60497700

CONTROL DATA CORPORATION

PASCAL VERSION 1 REFERENCE MANUAL

CDC® OPERATING SYSTEM:
NOS 2

REVISION RECORD

Revision	Description
01 (12/01/82)	Preliminary release at PSR level 580.
A (09/30/83)	This revision documents Pascal Version 1.1 at PSR level 596. Pascal Version 1.1 supersedes Pascal Version 1.0.

REVISION LETTERS I, O, Q, AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
P. O. BOX 3492
SUNNYVALE, CALIFORNIA 94088-3492

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Front Cover	-
Title Page	-
ii	A
iii/iv	A
v/vi	A
vii	A
viii	A
ix	A
1-1	A
2-1 thru 2-16	A
3-1 thru 3-7	A
4-1 thru 4-15	A
5-1 thru 5-22	A
6-1 thru 6-8	A
7-1 thru 7-11	A
8-1 thru 8-6	A
A-1 thru A-3	A
B-1 thru B-4	A
C-1 thru C-10	A
D-1 thru D-7	A
Index-1 thru -3	A
Comment Sheet/mailer	A
Back Cover	-

PREFACE

This manual describes the CONTROL DATA® Pascal Version 1.1 language. It is intended to be used as a reference, not as a tutorial, for users who are familiar with a version of Pascal.

Pascal Version 1.1 is available under control of the NOS 2 operating system on the CDC® CYBER 170 Series; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems.

Pascal Version 1.1 supersedes Pascal Version 1.0.

This manual is organized as follows:

Section 1 provides a description of the Pascal Version 1.1 extensions to the International Standards Organization (ISO) standard Pascal.

Section 2 describes language elements.

Section 3 describes the program heading.

Section 4 describes data declaration and definition.

Section 5 describes routines.

Section 6 describes statements.

Section 7 describes compiling, loading, and executing a Pascal program under the NOS 2 operating system.

Section 8 shows some complete Pascal programs.

Appendix A describes available character sets.

Appendix B describes compilation error messages.

Appendix C lists glossary definitions.

Appendix D describes the differences between Pascal Version 1.0 and Pascal Version 1.1.

Related material is contained in the NOS Version 2 Reference Set Volume 3, System Commands, publication number 60459680.

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

NOTATIONS

Certain notations are used throughout the manual with consistent meaning. These notations are:

--->	Indicates the permissible direction of traversal.
( )	Contains a reserved word or symbol in a syntax diagram. Alphabetic characters must appear in uppercase in your source program. A list of reserved words and symbols can be found in section 2.
[ ]	Contains the general name of a construct that you must further define; refer to the syntax diagram of the named item for definition rules.
UPPERCASE	Indicates a reserved word in the text. Alphabetic characters must appear in uppercase in your source code. A list of reserved words can be found in section 2.
. or	Indicates statements that are not shown and are not relevant to the example.
(CDC)	Indicates a Control Data extension to ISO standard Pascal.

All program statements in this manual are shown in the internal Pascal character set representation. You can translate special characters into the character set used at your site by referring to appendix A, Character Sets.

PASCAL VERSION 1.1 EXTENSIONS TO STANDARD PASCAL

1

Pascal Version 1.1 conforms to the International Standards Organization (ISO) standard Pascal. The following paragraphs introduce the Pascal Version 1.1 extensions to standard Pascal.

DATA DECLARATION AND DEFINITION

Pascal Version 1.1 has an additional predefined type called ALFA, which is associated with PACKED ARRAY[1..10] OF CHAR.

Pascal Version 1.1 has an additional data declaration section called the VALUE section. See section 4 for information about the VALUE section.

EXTERNAL DIRECTIVES

Pascal Version 1.1 allows access to external subroutines through the EXTERN and FORTRAN directives. See section 5 for information about the EXTERN and FORTRAN directives.

PREDEFINED ROUTINES

Pascal Version 1.1 has the following predefined procedures:

```
GETSEG(f,n)
PUTSEG(f,n)
REWRITE(f,n)
```

Pascal Version 1.1 has the following predefined functions:

```
CARD(a)    EOS(f)     MESSAGE(a)
CLOCK      EXPO(a)    TIME(a)
DATE(a)    HALT       TRUNC(a,n)
```

See section 5 for information about these predefined routines.

SEGMENTED FILE OPERATIONS

Pascal Version 1.1 allows segmented file operations. See section 3 for information about segmented files in the program heading, section 4 for information about segmented files in the data declaration and definition part, and section 5 for information about segmented files in predefined procedures.

STATEMENTS

Pascal Version 1.1 has an additional clause in the CASE statement called the OTHERWISE clause. See section 6 for information about the OTHERWISE clause in the CASE statement.

LANGUAGE ELEMENTS 2

This section describes the language elements of Pascal. The language elements are Pascal symbols, expressions, and separators. These elements are combined to form statements, which are the basic constructs of a program. Statements are described in section 6. Statements are grouped into blocks. This section also discusses blocks and the scope of identifiers within blocks.

PASCAL SYMBOLS

The basic language elements are Pascal symbols. Pascal symbols are analogous to the alphabet of the English language. The set of Pascal symbols is divided into six categories: reserved words and symbols, identifiers, numbers, labels, literals, and directives.

RESERVED WORDS AND SYMBOLS

A reserved word is a name that has an associated value in the Pascal language that cannot be changed. In this manual, reserved words are shown in underlined uppercase letters. The following is a list of the reserved words:

AND	FILE	OF	SET
ARRAY	FOR	OR	THEN
BEGIN	FUNCTION	OTHERWISE (CDC)	TO
CASE	GOTO	PACKED	TYPE
CONST	IF	PROCEDURE	UNTIL
DIV	IN	PROGRAM	VALUE (CDC)
DO	LABEL	RECORD	VAR
DOWNTO	MOD	REPEAT	WHILE
ELSE	NIL	SEGMENTED (CDC)	WITH
END	NOT		

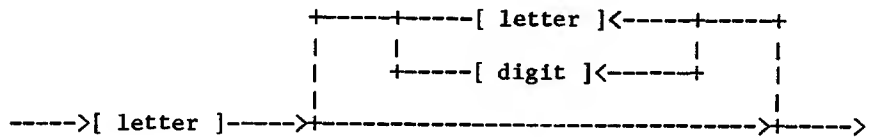
A reserved symbol is a symbol that has an associated value in the Pascal language that cannot be changed. Reserved words and symbols are used to delimit the syntactic units of your program. The following is a list of the reserved symbols:

+	,	<	[
-	;	<=	]
*	:	>=	(space)
/	'	>	..
:=	=	(	(*
.	<>	)	*)

IDENTIFIERS

An identifier is a name that denotes a constant, type, variable, value, procedure, or function. An identifier must begin with a letter followed by any combination of letters and digits up to 120 significant letters and digits. A letter must be an uppercase letter in the set A through Z. A digit must be in the set 0 through 9.

Identifier



The following are examples of valid and invalid identifiers:

Valid	Invalid	
SUITS	1A	
DAY	#SPACE	(where # is an initial space)
NO1	SALES#TAX	(where # is an embedded space)

The Post Mortem Dump (PMD) routine and the loader acknowledge only the first part of an identifier as significant. PMD acknowledges only the first ten characters of an identifier as significant. The loader acknowledges only the first seven characters of an entry-point identifier that is passed through the E compiler option as significant.

You cannot use a reserved word as an identifier. For example, you cannot redefine NIL to TRUE in your program.

You can use a predefined word as an identifier. A predefined word is an identifier that has an associated value in the Pascal language that can be changed. For example, you can redefine ALFA to ARRAY[1..10] OF INTEGER in your program. Table 2-1 shows the predefined words and their associated values.

TABLE 2-1. PREDEFINED WORDS

Predefined Word	Associated Value
ABS(a)	Returns the absolute value of a.
ALFA (CDC)	PACKED ARRAY[1..10] OF CHAR.
ARCTAN(a)	Returns the arctangent of a.
BOOLEAN	The predefined identifiers TRUE and FALSE.
CARD(a) (CDC)	Returns the cardinality of a.
CHAR	The character set used at your installation.
CHR(a)	Returns the character that has ordinal number a.
CLOCK (CDC)	Returns the current used CPU-time in milliseconds.
COS(a)	Returns the cosine of a.
DATE(a) (CDC)	Assigns the current date to a.
DISPOSE(p)	Releases the variable that is referenced by p. Any pointer that points to the variable referenced by p becomes undefined and the pointer itself becomes inaccessible.
EOF(f)	Returns a TRUE value if the end-of-file mark has been reached while reading file f or a FALSE value if the end-of-file mark has not been reached. If a file is not specified, INPUT is assumed.

(Continued on next page)

TABLE 2-1. PREDEFINED WORDS

(Continued)

Predefined Word	Associated Value
EOLN(f)	Returns a TRUE value if the end-of-line mark has been reached while reading file f or a FALSE value if the end-of-line mark has not been reached. If a file is not specified, INPUT is assumed.
EOS(f)	Returns a TRUE value if the end-of-segment mark has been reached while reading file f or a FALSE value if the end-of-segment mark has not been reached. If a file is not specified, INPUT is assumed.
EXP(a)	Returns the value of E(a).
EXPO(a) (CDC)	Returns the value of E(a) in binary representation.
FALSE	Zero.
GET(f)	Advances the position of the file to the next component and places the value of the component into the buffer variable f↑.
GETSEG(f[,n]) (CDC)	Begins reading at the beginning of the nth segment counting from the current position in file f. GETSEG(f,1) is equivalent to GETSEG(f).
HALT(a) (CDC)	Terminates the program, writes the argument in the dayfile of the job, and produces a dump.
INPUT	TEXT.
INTEGER	The range of decimal values [-2**48 + 1 .. 2**48 - 1], which is equivalent to the range of octal values [-7777777777777777 .. 7777777777777777].
LN(a)	Returns the value of the natural logarithm of a.
MAXINT	2**48 - 1.
MESSAGE(a) (CDC)	Writes a in the dayfile of the job.
NEW(p)	Allocates a new variable and assigns a reference to it.
ODD(a)	Returns a TRUE value if a is odd and a FALSE value if a is even.
ORD(a)	Returns the position of a in the set of values defined by the type of a.
OUTPUT	TEXT.

(Continued on next page)

TABLE 2-1. PREDEFINED WORDS

Predefined Word	Associated Value
PACK(a,i,z)	Takes the elements of array a beginning at subscript position i and copies them into packed array z beginning at the first subscript position.
PAGE(f)	Positions the line printer at the top of a new page before printing the next line in file f.
PRED(a)	Returns the predecessor of a; a cannot be REAL type. If a does not exist an error will occur.
PUT(f)	Appends the value of file buffer variable f↑ to file f.
PUTSEG(f[,n]) (CDC)	Closes the current segment of file f by putting end-of-segment mark.
READ(f,v[,v ...])	Positions file f and gets the referenced variables. If a file is not specified, INPUT is assumed.
READLN(f,v[,v ...])	Gets the referenced variables from file f. When a READLN is completed, any remaining values on the current input line, including an end-of-line, are discarded. The first value on the next line in file f will be read next.
REAL	The range of values [-10**322 .. -10**-293, 0, 10**-293 .. 10**322].
RESET(f)	Positions file f to the beginning-of-information. RESET(f) must be done on every input file except INPUT.
REWRITE(f[,n]) (CDC)	Empties file f and allows it to be written to. REWRITE(f) must be done on every output file except OUTPUT.
ROUND(a)	Returns a rounded to the nearest integer.
SEGMENTED TEXT (CDC)	SEGMENTED FILE OF TEXT.
SIN(a)	Returns the sine of a.
SQR(a)	Returns the square of a.
SQRT(a)	Returns the square root of a.
SUCC(a)	Returns the successor of a.
TEXT	FILE OF CHAR.

(Continued on next page)

TABLE 2-1. PREDEFINED WORDS

Predefined Word	Associated Value
TIME(a) (CDC)	Assigns the current time to a.
TRUE	One.
TRUNC(a[,n]) (CDC)	Returns either the largest integer $\leq$ a if a $\geq$ 0 or the smallest integer $\geq$ a if a $<$ 0.
UNDEFINED(a) (CDC)	Returns a TRUE value if a is out of range or indefinite or a FALSE value if a is not out of range or indefinite.
UNPACK(z,a,i)	Takes the elements beginning at the first subscript position of packed array z and copies them into array a beginning at subscript position i.
WRITE(f,e[,e ...])	Transforms the expressions into a sequence of characters and puts the sequence onto file f. If a file is not specified, OUTPUT is assumed.
WRITELN(f,e [,e ...])	The procedure that terminates the current line in file f by putting an end-of-line mark. If a file is not specified, OUTPUT is assumed.

See section 5 for more information about the routines that are associated with predefined words.

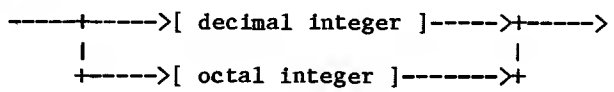
NUMBERS

A number can be either a signed or unsigned integer or a signed or unsigned real.

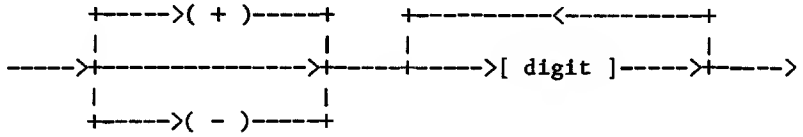
Integer Numbers

An integer number is a decimal or octal integer. A decimal integer is a signed integer in the range [-2**48 + 1 .. 2**48 - 1]. An octal integer is a signed integer in the range [-7777777777777777 .. 7777777777777777] and must be followed by the character B. The digits in an octal integer must be in the set 0 through 7.

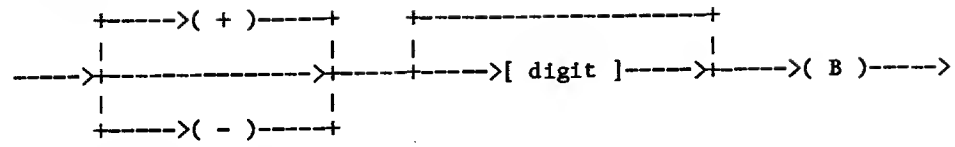
Integer



Decimal Integer



Octal Integer



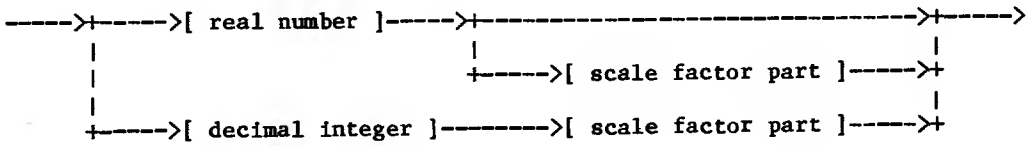
The following are examples of integer numbers:

```
-714      777B
```

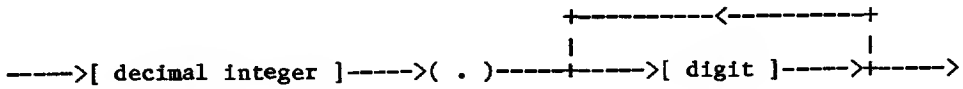
Real Numbers

A real number is either a real number with an optional scale factor or a decimal integer with a scale factor. A real number is a decimal integer followed by a decimal point and up to 14 integers. A real number must be in the range [-10**322 .. -10**-293, 0, 10**-293 .. 10**322]. A decimal integer is a signed integer in the range [-2**48 + 1 .. 2**48 - 1]. The letter E preceding a scale factor means times ten to the power of.

Real

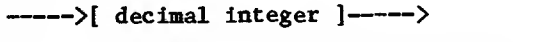


Real Number



Scale Factor Part

Scale Factor



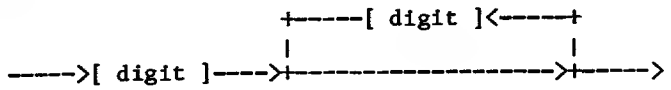
The following are examples of real numbers:

```
-3.14      0.314E1      314.0E-2
```

LABELS

A label is an unsigned integer in the range [0..9999].

Label



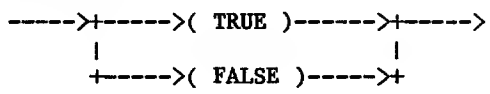
LITERALS

A literal is a symbol that holds a value. There are three kinds of literals: Boolean, character, and string.

Boolean Literals

A Boolean literal is either of the predefined words TRUE or FALSE.

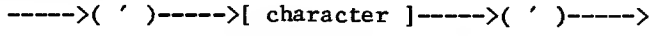
Boolean Literal



Character Literals

A character literal is a single character enclosed in single quote (') symbols.

Character Literal



The following are examples of character literals:

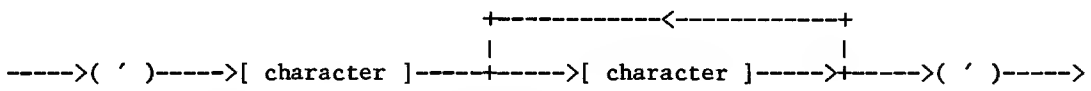
```
'C'     '+'     ''''
```

Note that inside a character literal a single quote (') symbol is denoted by two single quote ('') symbols.

String Literals

A string literal is a sequence of characters enclosed in single quote (') symbols.

String Literal



The following are examples of string literals:

```
'EQUALS'      '''LOOKLIKE'''
```

Note that inside a string literal a single quote (') symbol is denoted by two quote ('') symbols.

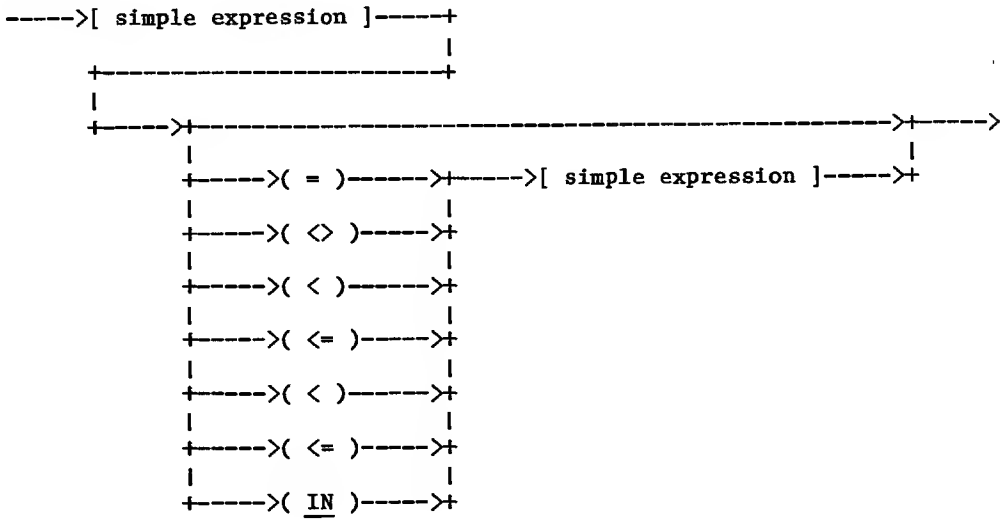
DIRECTIVES

A directive is an instruction in a procedure or function declaration that tells where the procedure or function declaration occurs in relation to the program block. There are three directives: FORWARD, FORTRAN (CDC), and EXTERN (CDC). See section 5 for information about these directives.

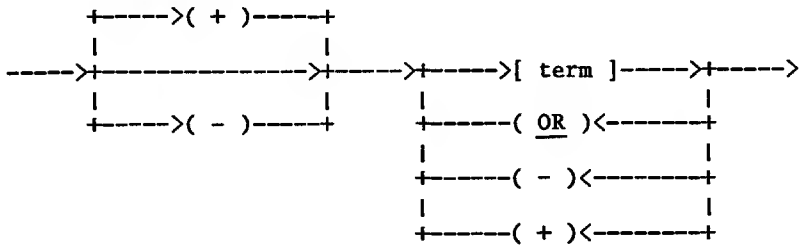
EXPRESSIONS

Expressions are combinations of Pascal symbols. Expressions are analogous to clauses in the English language. An expression defines a rule of computation from which a value can be obtained.

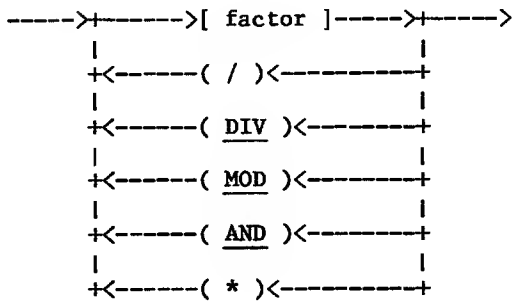
Expression



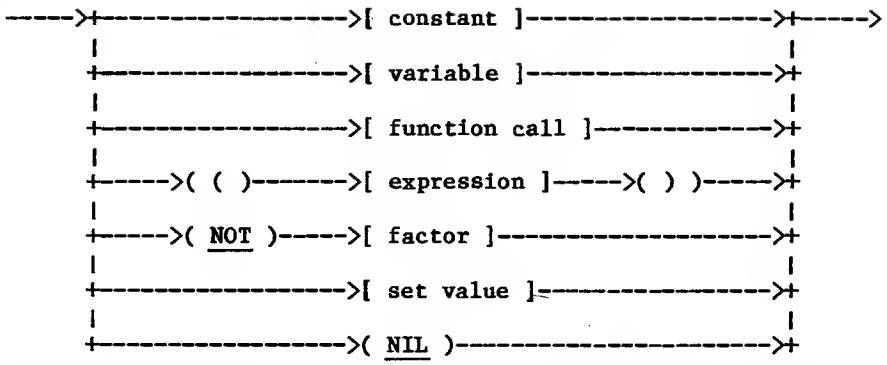
Simple Expression



Term



Factor



Expressions are written in infix notation, which means that a dyadic operator applies to the operands on either side of it. An expression is evaluated from left to right using the following precedence rules:

NOT	Highest precedence
*, /, DIV, MOD, AND	↓
+, -, OR	↓
<, >, =, <=, >=, <>, IN	Lowest precedence

For example, the following expressions on the left-hand side of the equal sign (=) are equivalent to the corresponding expressions on the right-hand side:

```
2 * 3 - 4 * 5 = (2 * 3) - (4 * 5)
15 DIV 4 * 4 = (15 DIV 4) * 4
80 / 5 / 3 = (80 / 5) / 3
4 + 2 * 3 = 4 + (2 * 3)
```

Likewise, for any B1, B2, B3 of type Boolean, the following expression on the left-hand side is equivalent to the expression on the right-hand side:

```
B1 OR NOT B2 AND B3 = B1 OR ((NOT B2) AND B3)
```

All factors in an expression are evaluated and should be defined.

If an expression contains a function whose evaluation causes side effects on other factors in the expression, the left to right evaluation does not always hold; such side effects should be avoided.

During evaluation of an expression, intermediate results are kept in a fixed number of registers. If the number of intermediate results exceeds the capacity of the registers, the expression cannot be translated and the compiler issues the error message: EXPRESSION TOO COMPLICATED. To remedy this, you must either rewrite the expression with a less complicated parenthesis structure or split the expression into two or more expressions.

OPERATORS

There are four kinds of operators: arithmetic, Boolean, relational, and set. Arithmetic operators are either dyadic or monadic. A dyadic operator operates on two operands, the operands on either side of the operator. A monadic operator operates on one operand, the operand on the right of the operator. Boolean, relational, and set operators are dyadic operators. Tables 2-2, 2-3, 2-4, 2-5, and 2-6 describe the operators. Any combination of operator types that is not listed in the tables will result in a compile-time error message.

TABLE 2-2. DYADIC ARITHMETIC OPERATORS

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
+	Addition	INTEGER INTEGER REAL REAL	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL
-	Subtraction	INTEGER INTEGER REAL REAL	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL
*	Multiplication	INTEGER INTEGER REAL REAL	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL
/	Division	INTEGER INTEGER REAL REAL	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL
DIV	Division with truncation	INTEGER	INTEGER	INTEGER
MOD	Modulo	INTEGER	INTEGER	INTEGER

TABLE 2-3. MONADIC ARITHMETIC OPERATORS

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
+	Identity	INTEGER INTEGER REAL REAL	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL
-	Sign inversion	INTEGER INTEGER REAL REAL	INTEGER REAL INTEGER REAL	INTEGER REAL REAL REAL

TABLE 2-4. BOOLEAN OPERATORS

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
AND	Logical conjunction	BOOLEAN	BOOLEAN	BOOLEAN
OR	Logical disjunction	BOOLEAN	BOOLEAN	BOOLEAN
NOT	Logical negation	BOOLEAN	BOOLEAN	BOOLEAN

TABLE 2-5. SET OPERATORS

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
+	Set union	SET OF T	SET OF T	SET OF T
-	Set difference	SET OF T	SET OF T	SET OF T
*	Set intersection	SET OF T	SET OF T	SET OF T

TABLE 2-6. RELATIONAL OPERATORS

Operator	Operation	Left Operand Type	Right Operand Type	Result Type
=	Logical comparison equal to	Simple type, string type, pointer type	Compatible type	BOOLEAN
		SET OF T	SET OF T	BOOLEAN
		INTEGER	REAL	BOOLEAN
		REAL	INTEGER	BOOLEAN
<>	Logical comparison not equal to	Simple type, string type, pointer type	Compatible type	BOOLEAN
		SET OF T	SET OF T	BOOLEAN
		INTEGER	REAL	BOOLEAN
		REAL	INTEGER	BOOLEAN
<	Logical comparison less than	Simple type, string type	Compatible type	BOOLEAN
		INTEGER	REAL	BOOLEAN
		REAL	INTEGER	BOOLEAN
>	Logical comparison greater than	Simple type, string type	Compatible type	BOOLEAN
		INTEGER	REAL	BOOLEAN
		REAL	INTEGER	BOOLEAN
<=	Logical comparison less than or equal to	Simple type, string type,	Compatible type	BOOLEAN
		SET OF T	SET OF T	BOOLEAN
		INTEGER	REAL	BOOLEAN
		REAL	INTEGER	BOOLEAN
>=	Logical comparison greater than or equal to	Simple type, string type,	Compatible type	BOOLEAN
		SET OF T	SET OF T	BOOLEAN
		INTEGER	REAL	BOOLEAN
		REAL	INTEGER	BOOLEAN
IN	Set membership	Scalar type T	SET OF T	BOOLEAN

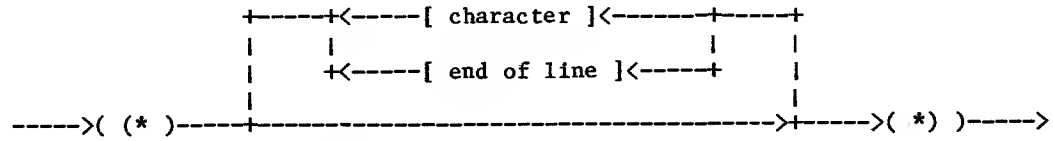
SEPARATORS

A separator is a comment or nonprinting symbol. Separators are analogous to commas in the English language. Zero or more separators can occur either between any two consecutive Pascal symbols or before the first Pascal symbol in a program block. A separator must appear between any two consecutive reserved words, identifiers, numbers, or labels. A space or a separator must appear between a numeric constant and a reserved word. For example, 5DIV J is invalid. The valid expression is 5 DIV J. A separator cannot occur within a Pascal symbol.

COMMENT

A comment is a string of explanatory text enclosed in an open comment symbol (* and a close comment symbol *). You can improve the readability of your program by adding comments, without affecting the results produced by the program.

Comment



Nested comments are not allowed.

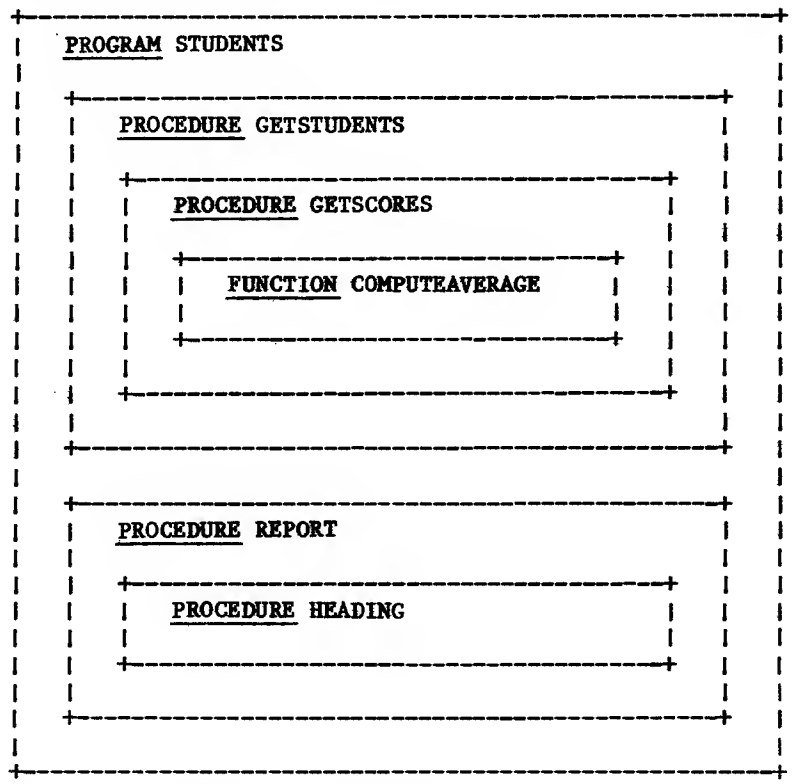
If the first character after the open comment symbol is a dollar sign ($), the comment is interpreted as a list of compiler options. See section 7 for a description of available compiler options.

NONPRINTING SYMBOL

A nonprinting symbol is either a space or the end-of-line mark.

BLOCKS

A block is a group of statements that is preceded by a PROGRAM, PROCEDURE, or FUNCTION heading. One block or many blocks can be nested inside a block. The following diagram shows some boxes that represent blocks in a program:



The part of a program over which an identifier is valid is called the scope of the identifier. The identifiers in each block are valid in the box in which they are declared and in any box that the block contains.

An identifier is valid throughout the entire program if it is declared in the CONST, TYPE, VAR, or VALUE section that follows the PROGRAM statement. An identifier that is valid throughout the entire program is called a global variable. For example, the following identifiers are global variables:

```
PROGRAM STUDENTS(CLASSFL,OUTPUT);
CONST
    NCLASSES = 3;
TYPE
    CLASS = RECORD
              CLASSNAME : PACKED ARRAY[1..14] OF CHAR;
              PLACING : ARRAY[1..4] OF INTEGER;
              CUTS : ARRAY[1..3] OF INTEGER;
              SCORE : INTEGER
            END;
VAR
    CLASSFL : TEXT;
PROCEDURE GETSTUDENTS;
```

The identifier GETSTUDENTS is also a global variable; however, the variables that are declared in the declaration and definition section that follows the PROCEDURE heading are valid only within the procedure. An identifier that appears in the CONST, TYPE, or VAR section that follows the PROCEDURE heading is called a local variable because it is valid only within the procedure.

The variables that are declared within a nested procedure or function definition are local to the procedure or function. For example, the variables M and N are local to procedure HEADING; however, the variables MAXLINES, PAGE, and LINES are available to procedure HEADING:

```
PROCEDURE REPORT;
CONST
    MAXLINES = 60;
VAR
    PAGE, LINES : INTEGER;
PROCEDURE HEADING;
VAR
    M, N : INTEGER;
```

Every identifier that is not a reserved word or a predefined word must be declared or defined before it is used, with one exception. The exception is that a pointer identifier can be declared before the pointer is defined. For example, in the following sequence, the declaration POINTER = ↑TEAMDEFN can occur before the definition of TEAMDEFN as long as both declarations occur in the same TYPE section:

```
TYPE
    POINTER = ↑TEAMDEFN;
    TEAMDEFN =  RECORD
                    NUMBER : INTEGER;
                    TEAMNAME : PACKED ARRAY[1..20] OF CHAR;
                    TEAMMEMBERS : ARRAY[1..3] OF MEMBER;
                    NEXTPNTR : POINTER
                END; (* RECORD *)
```

SCOPE RULES

A scope is the portion of a program block over which an identifier is valid. A scope is defined by one of the following:

A field list (excluding inner scopes)

A routine heading

A block (excluding inner scopes)

You can declare an identifier only once in each scope. Therefore, the following sequence is invalid:

```
CONST
    BIG = MAXINT;     <----- MAXINT is predefined
    MAXINT = 0;       <----- MAXINT is redefined
```

If an identifier is declared both in a scope definition and in an inner scope, then the inner scope declaration is effective in the inner scope.

An identifier in the innermost scope of a triply nested scope (a scope within a scope within the scope that defines the identifier) cannot be redefined in the next-to-innermost scope. For example, in the following sequence there is a conflict between the use of T in scope C and the use of T in scope B. Therefore, the program sequence is not a valid TYPE definition section.

```
TYPE                                    ┐
    R = RECORD                          ┐   |
            S : RECORD          ┐       |   |
                  X : ↑T        | C     |   |
                END;            ┘       | B |  A
            T : INTEGER                 |   |
        END;                            ┘   |
    T = REAL;                               ┘
```

Generally, the declaration of an identifier is effective in the rest of the block where it is declared. However, details for each kind of identifier are given below.

- Constant identifier, type identifier, variable identifier, enumeration constant, label, and routine identifier:

  The declaration of these identifiers is effective in the rest of the block.

- Pointer type identifier:

  In the definition of a pointer type, the type identifier on the right-hand side of the arrow (↑) can be defined textually after the pointer type definition.

- Field identifier:

  The declaration of a field identifier is effective in the rest of the block, but the field identifier can be used only in RECORD variables and WITH statements.

- Routine parameter name:

  The name is effective in the rest of the block.

- Program identifier:

  The program identifier has no significance within the program.

- Parameter identifier:

  The identifier extends over the entire parameter list in which it is declared and also over the block of the procedure or function that corresponds to the parameter list. However, the scope does not include the procedure or function name or the result-type identifier in the function. For example, in the following function, the use of G as a formal parameter in the function heading does not conflict with the use of G as the function result identifier:

```
FUNCTION F (G : INTEGER) : G;
BEGIN
    PROC(G);
    F := SOMETHING
END;
```

  In the following example, the use of P as a formal parameter does not conflict with the use of P as the procedure name. However, the use of P as a formal parameter does prevent the procedure from calling itself because a P inside the procedure block can refer only to the parameter.

```
PROCEDURE P (VAR P : INTEGER);
BEGIN
    P := P + 1
END;
```

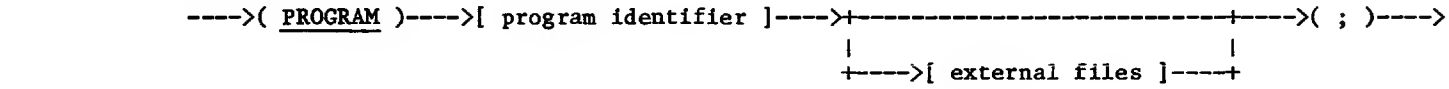
PROGRAM HEADING 3

A Pascal program must contain a program heading, data declarations and definitions, and a statements block. This section describes the program heading. A program heading consists of a PROGRAM statement.

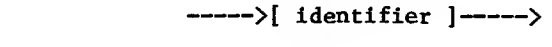
PROGRAM STATEMENT

The PROGRAM statement identifies the program to the operating system and contains a list of the external files that are used in the program.

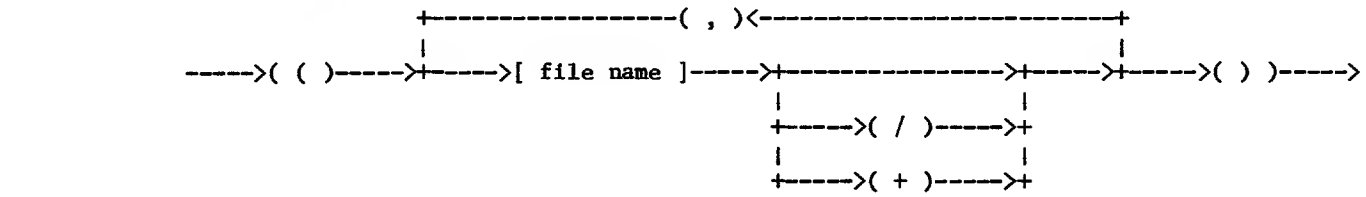
Program Statement



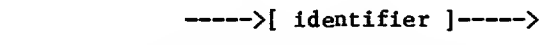
Program Identifier



External File List



File Name



A program identifier cannot be a reserved word. An external file identifier can contain at most 7 characters.

EXTERNAL FILE LIST WITH PREDEFINED FILES INPUT AND OUTPUT

The predefined files INPUT and OUTPUT are textfiles. A textfile is a file of characters that can be divided into lines. The mark that indicates the end-of-line is not a character in the character set; end-of-line is a separator whose value can be tested with the predefined function EOLN. See section 5 for more information about the predefined function EOLN.

Textfiles INPUT and OUTPUT correspond to NOS files INPUT and OUTPUT, respectively.

You can input data from the default external input file (INPUT) or output data to the default external output file (OUTPUT) by specifying INPUT or OUTPUT as parameters on the PROGRAM statement. For example,

```
PROGRAM STUDENTS(INPUT,OUTPUT);
```

When you specify INPUT or OUTPUT as parameters on the PROGRAM statement, you do not have to declare the type of the file or position the file through statements within the program because INPUT and OUTPUT are implicitly declared and positioned as follows:

```
VAR
    INPUT, OUTPUT : TEXT;
BEGIN
    RESET(INPUT);
    REWRITE(OUTPUT);
```

When you use INPUT or OUTPUT on the PROGRAM statement, you can use either of the following compile and execution command pairs:

```
PASCAL (I=source-file-name, B=object-file-name [/options])
object-file-name

PASCAL (I=source-file-name [/options])
LGO [(external-file-name [,external-file-name ...])]
```

where LGO is the default object-file-name and [ ] encloses optional parts of the commands. The compile command options are described in section 7.

The operating system associates the external files specified on the execute command with the external files on the PROGRAM statement by position. Table 3-1 shows examples of how the operating system associates external files on the execute command with predefined external files on the PROGRAM statement.

TABLE 3-1. EXTERNAL FILE ASSOCIATION WITH PREDEFINED FILES

Given	Association
LGO (INDATA,OUTDATA) PROGRAM STUDENTS(INPUT,OUTPUT);	External files on the execute command and INPUT and OUTPUT on the PROGRAM statement will cause INDATA to be used as the input file and OUTDATA to be used as the output file. INDATA and OUTDATA must be defined and positioned through statements in your program.
LGO (INDATA) PROGRAM STUDENTS(INPUT,OUTPUT);	Will cause INDATA to be used as the input file and OUTPUT to be used as the output file.
LGO (,OUTDATA) PROGRAM STUDENTS(INPUT,OUTPUT);	Will cause INPUT to be used as the input file and OUTDATA to be used as the output file.
LGO (INDATA,OUTDATA) PROGRAM STUDENTS;	External files on the execute command and no external files on the PROGRAM statement will cause an error when INDATA and OUTDATA are defined in your program.
LGO PROGRAM STUDENTS(INPUT,OUTPUT);	No external files on the execute command and INPUT and OUTPUT on the PROGRAM statement will cause the default external files to be used.
LGO PROGRAM STUDENTS;	No external files on the execute command and no external files on the PROGRAM statement will cause an error if a file operation is attempted.

EXTERNAL FILE LIST WITH USER-DEFINED FILES

You can input data from user-defined external input files or output data to user-defined external output files by specifying them as parameters on the PROGRAM statement. For example,

```
PROGRAM STUDENTS(INDATA,OUTDATA);
```

When you specify user-defined files as parameters on the PROGRAM statement, you must declare the type of the file and position the file through statements within the program. For example, INDATA and OUTDATA must be declared and positioned as follows:

```
VAR
    INDATA, OUTDATA : TEXT;
BEGIN
    RESET(INDATA);
    REWRITE(OUTDATA);
```

You must specify either the default output file OUTPUT or a user-defined output file on the PROGRAM statement in order to do a Post Mortem Dump.

When you use user-defined files on the PROGRAM statement, you can use either of the following compile and execution command pairs:

```
PASCAL (I=source-file-name, B=object-file-name [/options])
object-file-name [ (external-file-name [,external-file-name ...]) ]

PASCAL (I=source-file-name [/options])
LGO [(external-file-name [,external-file-name ...])]
```

where LGO is the default object-file-name and [ ] encloses optional parts of the commands. The compile command options are described in section 7.

The operating system associates the files specified on the execute command with the external files specified on the PROGRAM statement by position. Table 3-2 shows examples of how the operating system associates external files on the execute command with user-defined external files on the PROGRAM statement.

TABLE 3-2. EXTERNAL FILE ASSOCIATION WITH USER-DEFINED FILES

Given	Association
LGO (INDATA,OUTDATA) PROGRAM STUDENTS(INPUT,OUTPUT);	External files on the execute command and INPUT and OUTPUT on the PROGRAM statement will cause INDATA to be used as the input file and OUTDATA to be used as the output file. INDATA and OUTDATA must be defined and positioned through statements in your program.
LGO (INDATA) PROGRAM STUDENTS(INPUT,OUTPUT);	Will cause INDATA to be used as the input file and OUTPUT to be used as the output file.
LGO (,OUTDATA) PROGRAM STUDENTS(INPUT,OUTPUT);	Will cause INPUT to be used as the input file and OUTDATA to be used as the output file.
LGO (,X) PROGRAM STUDENTS(INPUT,A,B);	Will cause X to be used when a file operation that involves A is performed in your program.
LGO (INDATA,OUTDATA) PROGRAM STUDENTS;	External files on the execute command and no external files on the PROGRAM statement will cause an error when INDATA and OUTDATA are defined in your program.
LGO PROGRAM STUDENTS(INDATA,OUTDATA);	No external files on the execute command and INDATA and OUTDATA on the PROGRAM statement will cause INDATA and OUTDATA to be used as they are defined in your program.
LGO PROGRAM STUDENTS;	No external files on the execute command and no external files on the PROGRAM statement will cause an error if a file operation is attempted.

EXTERNAL FILE LIST WITH NO FILES

If the external file list on the PROGRAM statement is empty, then the Pascal compiler does not scan the execution command for external file names. This means that the execution command can contain execution-time parameters. For example, the following execution command

```
LGO (RPTCNT=7)
```

directs the Pascal compiler to perform program execution seven times.

If the external file list is empty, then Post Mortem Dump (PMD) cannot be used. You must specify an output file on the PROGRAM statement in order to use PMD.

You can put execution-time parameters on the execution command even when there are external files listed on the PROGRAM statement by specifying a comma on the execution command for each external file listed on the PROGRAM statement. For example, in the following execution command and PROGRAM statement

```
LGO (,,RPTCNT=7)
PROGRAM STUDENTS(INPUT,OUTPUT);
```

INPUT and OUTPUT are matched by position with the omitted file names, then RPTCNT=7 is accepted as an execution-time parameter.

EXTERNAL FILE LIST WITH INTERACTIVE FILES

To use an external file as an interactive file, you must specify a slash (/) after the external file name on the PROGRAM statement. For example,

```
PROGRAM STUDENTS(INPUT/,OUTPUT);
```

An interactive file is treated differently in terms of when the buffers are flushed. When a READ or READLN statement is executed, the output buffer is flushed and then the read operation occurs. With a noninteractive file, the output buffer is flushed after the read operation occurs. The difference between interactive and noninteractive files is important when you want to request input from the terminal. For example, in figure 3-1, a WRITELN statement requests input and a READ statement prompts input. Execution of the program shows that the output buffer, which contains "enter a real number for r," is flushed after the input prompt is given.

```
    1 PROGRAM INTER(INPUT,OUTPUT);
    2 CONST
    3    PI = 3.14159;
    4 VAR
    5    R, AREA : REAL;
    6 BEGIN
    7    WRITELN('ENTER A REAL NUMBER FOR R');
    8    READ(R);
    9    AREA := PI * SQR(R);
   10    WRITELN('THE AREA OF THE CIRCLE WITH RADIUS ',R:5,' IS',AREA)
   11 END.

              COMPILER-ESTIMATED 'W' OPTION = 003100B.


      0.044 CP SECS,  47252B CM USED.
  /LGO
  ? 5.0
  ENTER A REAL NUMBER FOR R
  THE AREA OF THE CIRCLE WITH RADIUS  5.0E+000 IS   7.8539750000000E+001
      0.002 CP SECS,   5404B CM USED.
```

Figure 3-1. Program INTER With INPUT as a Noninteractive File

To correct the program so that the request for input is written before the input prompt, you must insert a slash after the external file INPUT in the PROGRAM statement, as shown in figure 3-2.

```
    1 PROGRAM INTER(INPUT/,OUTPUT);
    2 CONST
    3    PI = 3.14159;
    4 VAR
    5    R, AREA : REAL;
    6 BEGIN
    7    WRITELN('ENTER A REAL NUMBER FOR R');
    8    READ(R);
    9    AREA := PI * SQR(R);
   10    WRITELN('THE AREA OF THE CIRCLE WITH RADIUS ',R:5,' IS',AREA)
   11 END.

              COMPILER-ESTIMATED 'W' OPTION = 003100B.


    0.044 CP SECS,  47252B CM USED.
  /LGO
  ENTER A REAL NUMBER FOR R
  ? 5.0
  THE AREA OF THE CIRCLE WITH RADIUS  5.0E+000 IS   7.8539750000000E+001
      0.002 CP SECS,   5404B CM USED.
```

Figure 3-2. Program INTER With INPUT as an Interactive File

An alternate way to correct the program in figure 3-1 is to insert a READLN statement before the READ statement. The READLN statement flushes the output buffer so that the prompt for input appears after the request for input.

A runtime and disk space penalty is extracted when an interactive flag is set on a noninteractive input file. A heavy runtime and disk space penalty is extracted when an interactive flag is set on a noninteractive output file.

EXTERNAL FILE LIST WITH SEGMENTED FILES (CDC)

A segmented file is a file that is divided into segments of varying lengths. A Pascal segmented file is equivalent to a NOS multirecord file. One Pascal segment is equivalent to one NOS record or one Record Manager section. One advantage of a segmented file is that you can manipulate logical records within a segment. Another advantage is that you can read multirecord files, such as CCL procedure files and loader binary files. The Pascal statement GETSEG(f,n) is equivalent to the NOS commands SKIPR,f,n if n > 0 or BKSP,f,n+1 if n <= 0.

To use the predefined external files INPUT or OUTPUT as segmented files, you must specify a plus sign (+) after the external file name on the PROGRAM statement. For example,

```
PROGRAM STUDENTS(INPUT+,OUTPUT);
```

The plus sign changes the predefined declaration of INPUT to the following declaration:

```
VAR
    INPUT : SEGMENTED TEXT;
```

The predefined procedures RESET(f) or REWRITE(f[,n]) must be specified for each segmented file, except INPUT or OUTPUT, before the file is read from or written to. To read from or write to a segmented file, use the predefined procedures GETSEG(f[,n]) and PUTSEG(f). REWRITE(f[,n]) and GETSEG(f[,n]) accept two arguments because the second argument denotes the segment to be operated on. Segmented files are intended for sequential forward processing, which means that REWRITE(f[,n]) and GETSEG(f[,n]) operations are not as efficient for n <= 0 as they are for n > 0. A Boolean function, EOS(f), is available for end-of-segment testing. See section 5 for more information about predefined procedures and functions.

DATA DECLARATION AND DEFINITION 4

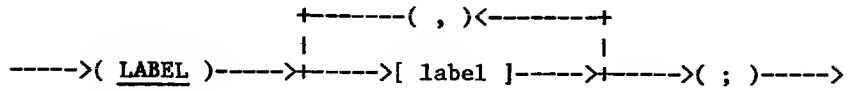
A Pascal program must contain a program heading, data declarations and definitions, and a statements block. This section describes data declarations and definitions.

Data declarations and definitions describe the data that will be manipulated in the statements block. Seven sections can appear in this part, although any of them may be empty. The section headings are: LABEL, CONST, TYPE, VAR, VALUE (CDC), PROCEDURE, and FUNCTION. The sections must appear in the listed order. PROCEDURE and FUNCTION declarations are discussed in section 5.

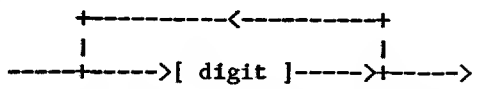
LABEL SECTION

The LABEL declaration section consists of definitions of numbers that will be used as statement labels in the statement part of the routine or program block. A label is an integer in the range [0..9999]. Two labels that denote the same number are considered identical.

Label Declaration Section



Label



The following is an example of the LABEL declaration section:

```
LABEL
    100, 200;
```

You must declare a label in the label declaration section of the routine or program block where it is defined. A declared label must be defined in the routine or program block. You define a label by prefixing an executable statement with the label and a colon (:). For example,

```
100 : A := SUCC(THURSDAY);
```

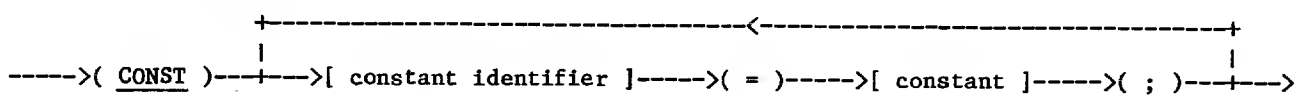
The statement after the label and colon cannot be a labeled statement.

You can define a label only once in the BEGIN/END block of the routine or program unit where it is declared.

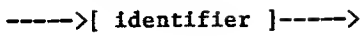
CONST SECTION

The CONST definition section consists of a number of definitions of constant identifiers. Each definition introduces an identifier as a synonym for the value of a literal or as a synonym for an enumeration constant from a scalar type.

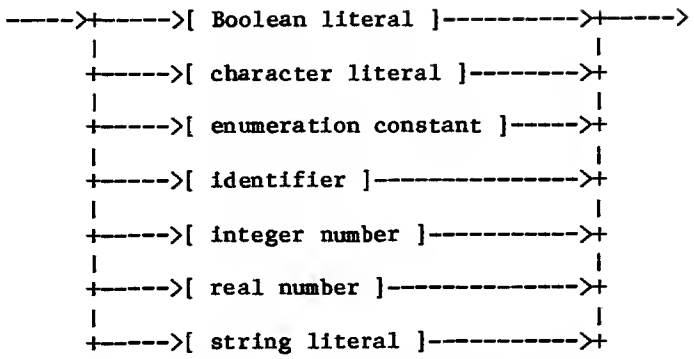
Constant Definition Section



Constant Identifier



Constant

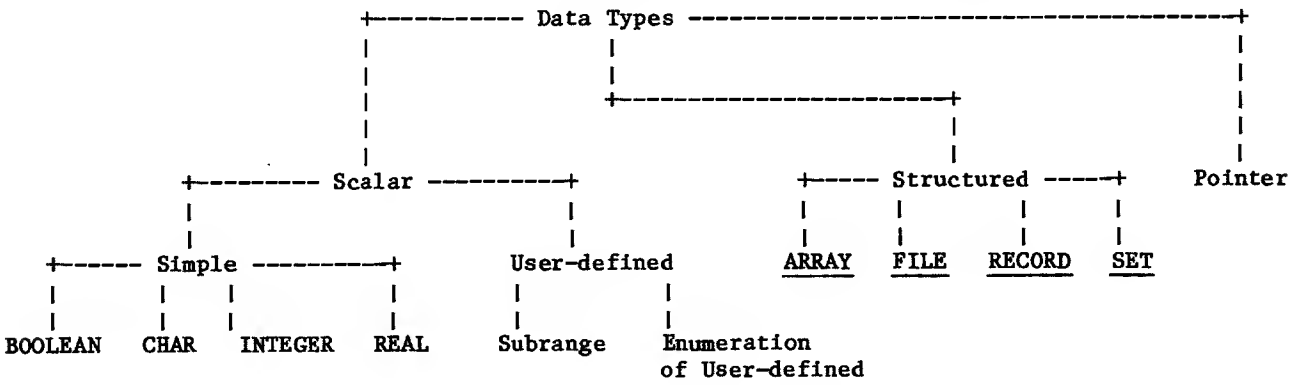


The following is an example of the CONST definition section:

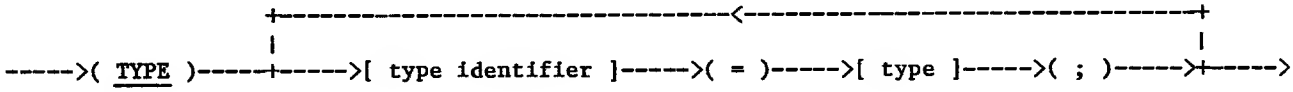
```
CONST
    UPPERLIMIT = 100;
    HEADING = 'TABLE PROGRAM N = 100';
```

TYPE SECTION

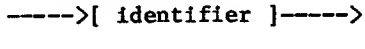
The TYPE declaration section defines sets of values that can be assumed by variables and expressions (operands) of that type. The following diagram shows the categories of Pascal data types:



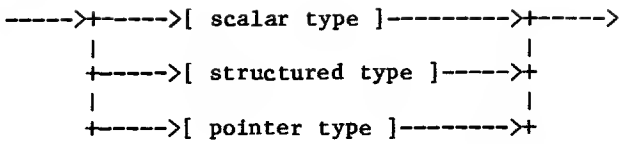
Type Definition Section



Type Identifier



Type



The following is an example of the TYPE definition section:

```
TYPE
    SUITS = (CLUB,DIAMOND,HEART,SPADE);
    DAYS = (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,
    FRIDAY,SATURDAY,SUNDAY);
    WEEKEND = FRIDAY..SUNDAY;
    MONTHS = (JANUARY,FEBRUARY,MARCH,APRIL,MAY,JUNE,JULY,AUGUST,
    SEPTEMBER,OCTOBER,NOVEMBER,DECEMBER);
    SEASONS = (WINTER,SPRING,SUMMER,AUTUMN);
    COLORS = (BLACK,RED);
```

Given the above TYPE definition section, the following relations are true:

```
DIAMOND <= HEART
MONDAY < SUNDAY
DECEMBER >= APRIL
WEDNESDAY = SUCC(TUESDAY)
NOVEMBER = PRED(DECEMBER)
```

The following relations are false:

```
CLUB >= DIAMOND
JANUARY = FEBRUARY
SUCC(NOVEMBER) = OCTOBER
```

The following expressions are undefined:

```
SUCC(SPADE)
PRED(MONDAY)
SUCC(DECEMBER)
```

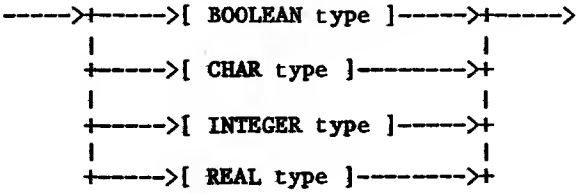
SCALAR DATA TYPES

Scalar data types are the fundamental data type. A scalar data type defines or enumerates all the possible values in an ordered set of data objects. Scalar data types are divided into two categories: simple and user-defined.

Simple Scalar Data Types

A simple scalar data type is an ordered set of data objects that is predefined in the Pascal language. There are four simple scalar data types: BOOLEAN, CHAR, INTEGER, and REAL.

Simple Scalar Type



BOOLEAN Type

BOOLEAN type is predefined as the ordered set [TRUE, FALSE], where TRUE > FALSE.

CHAR Type

CHAR type is predefined as the ordered set of characters used at your site. Possible character sets are the CDC Scientific (63 and 64) and the CDC ASCII (63 and 64). Appendix A shows the translations between the Pascal characters and the CDC Scientific and CDC ASCII characters.

The characters in any character set are numbered. The number that describes the position of the character within the set is called the ordinal number. The ordinal number can be obtained from the following Pascal character table by adding the row and column number for the character.

	0	1	2	3	4	5	6	7	8	9
0		A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	0	1	2
30	3	4	5	6	7	8	9	+	-	*
40	/	(	)	$	=		,	.	#	[
50	]	:	"	_	!	&	'	?	<	>
60	@	\	↑	;						

You can produce the table with the following program:

```
PROGRAM TABLE(OUTPUT);
VAR
    CHARACTER : CHAR;
BEGIN
    WRITELN('     0  1  2  3  4  5  6  7  8  9');
    WRITE(0:3,' ':3);
    FOR CHARACTER := 'A' TO ';' DO BEGIN
        IF (ORD(CHARACTER) MOD 10 = 0) THEN BEGIN
            WRITELN;
            WRITE(ORD(CHARACTER):3)
        END; (* IF *)
        WRITE(CHARACTER:3)
    END; (* FOR *)
END. (* TABLE *)
```

The character that precedes another character in the character set will always have a greater ordinal number. For example, this statement will always be true for any two characters C1 and C2:

```
(C1 < C2) = (ORD(C1) < ORD(C2))
```

INTEGER Type

INTEGER type is predefined as the ordered set [-2**48 + 1 .. 2**48 - 1] according to the natural ordering of integer numbers. Actually, integers in the range [-2**59 + 1 .. 2**59 - 1] can be stored, but the only operations that are executed correctly in the extended part of the range are: addition, subtraction, taking the absolute value, comparisons, and multiplication and division by certain constants. These constants must be either a power of two or the sum or difference of two powers of two.

REAL Type

REAL type is predefined as the ordered set [-10**322 .. -10**-293, 0, 10**-293 .. 10**322] according to the natural ordering of real numbers. A value of REAL type is represented in the CDC floating point format of a 48-bit mantissa and 11-bit sign and exponent; there are at least 14 significant decimal digits.

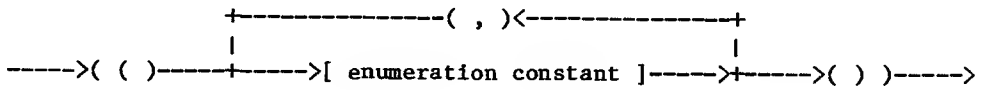
User-Defined Scalar Data Types

A user-defined scalar data type is an ordered set of data objects that are constructed from a subset of the simple scalar data types. There are two user-defined scalar data types: enumeration of a user-defined type and subrange type.

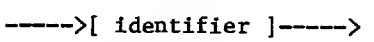
Enumeration of a User-Defined Type

Enumeration of a user-defined type is a set of user-defined constants.

Enumeration of a User-Defined Type



Enumeration Constant



The following are examples of enumeration of user-defined type definitions:

```
TYPE
    WEEKDAYS = (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY);
    WEEKENDS = (SATURDAY,SUNDAY);
    FACECARDS = (JACK,QUEEN,KING);
```

A constant that appears in one enumeration of a user-defined type definition cannot appear in another enumeration of a user-defined type definition because an expression that involves the common member would be ambiguous. For example, the definition DAYSOFF = (SATURDAY,SUNDAY,MONDAY); cannot appear in the TYPE section above because the expression SUNDAY > THURSDAY would be ambiguous.

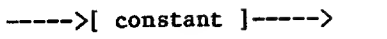
Subrange Type

Subrange type is a subset of either a simple scalar data type (except REAL type) or an enumeration of a user-defined type. The subset is defined by specifying minimum and maximum values separated by a double period (..). The minimum bound must not exceed the maximum bound, and the bounds must be of the same simple scalar data type.

Subrange Type

Minimum Bound

Maximum Bound



The following are examples of subrange type declarations:

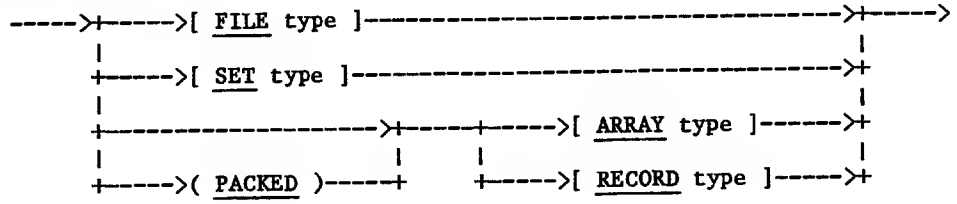
```
TYPE
    DAYS = (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY);
    WEEKEND = SATURDAY..SUNDAY;

TYPE
    CARDS = (ONE,TWO,THREE,FOUR,FIVE,SIX,SEVEN,EIGHT,NINE,TEN,
             JACK,QUEEN,KING);
    FACECARDS = JACK..KING;
```

STRUCTURED DATA TYPES

A structured type is composed of scalar types. The definition of a structured type specifies the structuring method and the component types.

There are four kinds of structured types: ARRAY, FILE, RECORD, and SET.

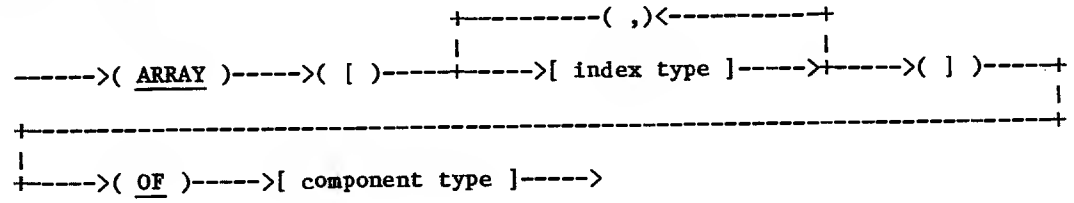
Structured Type



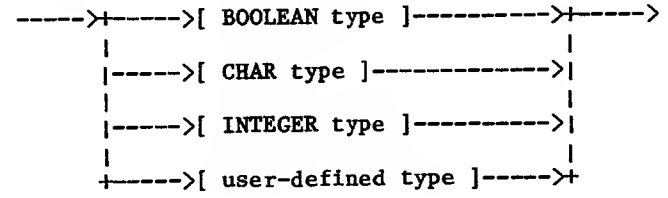
ARRAY Type

An array is a set of components that is identified by a single name. ARRAY type describes the set of components in the array. The number of components is specified by an enumeration type, which is called the index type.

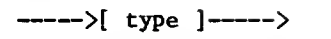
ARRAY Type



Index Type



Component Type



The index type is static and cannot be varied dynamically. This implies that the index type must be known at compilation time.

A shorthand notation for the type

```
ARRAY[T1] OF ARRAY[T2] OF T3
```

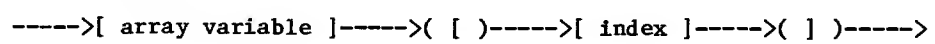
is the notation

```
ARRAY[T1,T2] OF T3
```

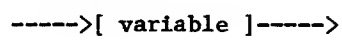
This is called a multidimensional array. The number of index types is called the dimension of the array. The array with index type T2 is called the innermost array.

You can use arrays either whole or by component. A whole array is selected by its array variable. A component of an array is selected by the array variable followed by an index enclosed in brackets. The total number of index expressions must not exceed the dimension of the array. Furthermore, the value of each index expression must be an enumeration type that is compatible with the corresponding index type.

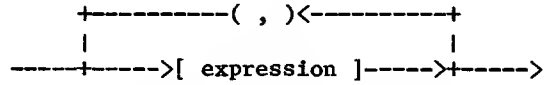
Indexed Variable



Array Variable



Index



The notations NAME[A1][A2] and NAME[A1,A2] can be used interchangeably.

The following are examples of array declarations:

```
TYPE
    HOURS = 8..16;
    MATRIX = ARRAY[1..N,1..N] OF REAL; (* N IS AN INTEGER CONSTANT *)
    COUNTER = ARRAY['A'..'Z'] OF INTEGER;
    NAMEOFDAY = ARRAY[DAYS] OF ALFA;
    OCCUPIED = ARRAY[DAYS,HOURS] OF BOOLEAN;
VAR
    A,B,C : MATRIX;
```

The following statements show array denotations:

```
A := B; (* THE ENTIRE MATRIX B IS COPIED INTO A *)
C[I] := A[I]; (* ONE ROW OF A IS COPIED INTO ONE ROW OF C *)
C[I,J] := A[K,L]; (* ONE COMPONENT OF A IS COPIED TO ONE COMPONENT
                     OF C *)
OCCUPIED[WEDNESDAY,9] := TRUE;
OCCUPIED[FRIDAY,15] := FALSE;
```

The following statements initialize B to the identity matrix:

```
FOR I := 1 TO N DO BEGIN
    FOR J := 1 TO N DO B[I,J] := 0;
    B[I,I] := 1
END; (* FOR *)
```

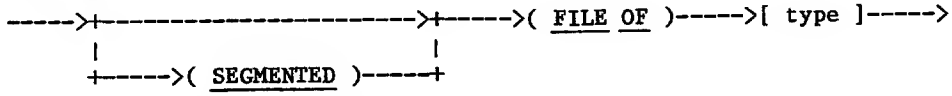
The following statements provide an alternative way of initializing B to the identity matrix:

```
FOR J := 1 TO N DO B[1][J] := 0;

FOR I := 2 TO N DO B[I] := B[1];

FOR I := 1 TO N DO B[I,I] := 1;
```

FILE Type

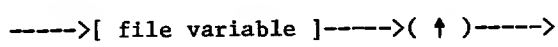
A file is a fixed number of like components that are called records. FILE type describes the records in the file. The declaration of a file variable introduces a file buffer to the component type. The file buffer is denoted by the file variable followed by an arrow (↑).

FILE Type

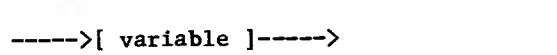


The file buffer is a template that can be positioned over any part of the file. The template isolates the part of the file that you want to read from or write to. The template is moved by certain file operations. The file operations cannot alternate between reading and writing; a file can be either read from or written to.

File Buffer



File Variable



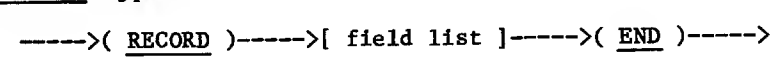
The sequential processing and the existence of a file buffer suggests that files are associated with secondary storage and peripherals. Exactly how the components are allocated varies. Usually only a few components are present in primary storage at a time, and only the component denoted by the file buffer is accessible.

A special mark is placed after the last component of the file. This mark is called the end-of-file (EOF) mark.

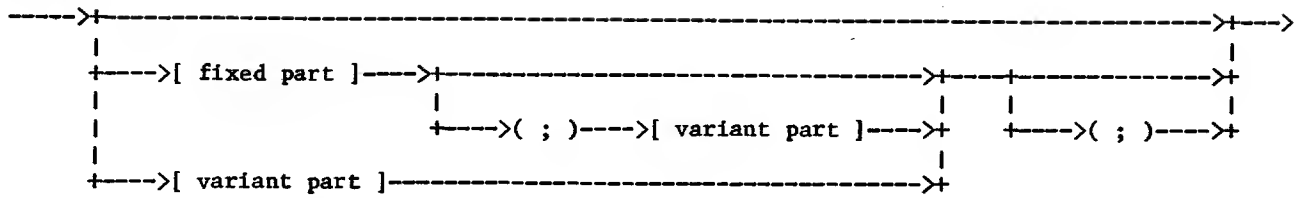
RECORD Type

A record is a fixed number of components called fields. RECORD type describes the fields in a record. A field list can be empty or can have a fixed part, a fixed part and a variant part, or a variant part.

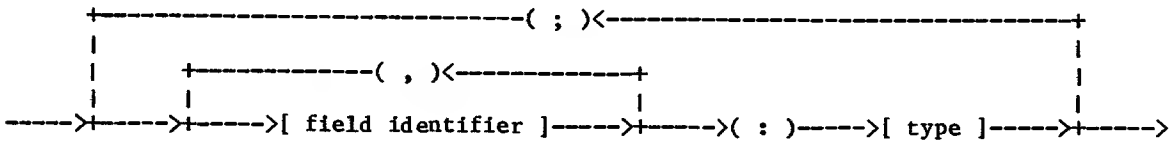
RECORD Type



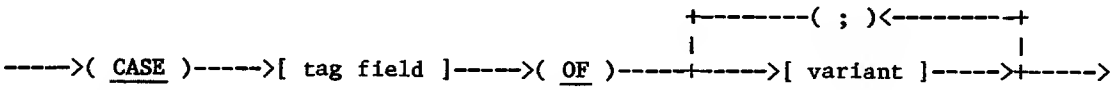
Field List



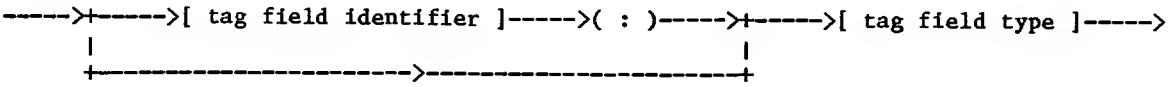
Fixed Part



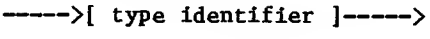
Variant Part



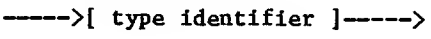
Tag Field



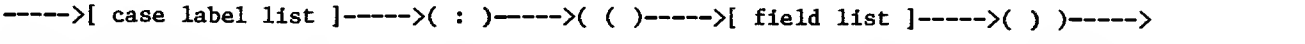
Tag Field Identifier



Tag Field Type



Variant



The variant part can have a number of variants. Multiple variants must have a tag field containing a value that indicates which variant is assumed by the field list at a given time. The tag field type can be an enumeration type, a FILE type, or a type that contains a FILE type. Every element of the enumeration type must be present in one of the tag fields. A tag field must be represented by a constant followed by a variant. All constants must be distinct. For example, RECORD CASE INTEGER OF ... cannot be used in practice because the tag field would require that each integer be used as a constant label; one solution to this problem is the following sequence:

```
TYPE
    TAGTYPE = 1..3;
    RECTYPE =   RECORD CASE TAG : TAGTYPE OF
                    1 : (FIELD1 : INTEGER);
                    2 : ();
                    3 : (FIELDX : REAL)
                END; (* RECORD *)
```

No constant that prefixes a variant can lie outside the range of its tag type. For example, given the tag type defined above, the following sequence is not valid because the constant 4 lies outside the range of the tag type TAGTYPE:

```
RECORD CASE TAGTYPE OF 1, 2, 3 : (); 4 : () END;
```

A tag field cannot be passed as an actual variable parameter. A tag field can be passed as a value parameter. A tag field can have the same spelling as a type identifier. For example, the following sequence is valid:

```
RECORD CASE INTEGER : BOOLEAN OF
    TRUE : ();
    FALSE : ()
END; (* RECORD *)
```

A record can be used either as a whole or as a component. A component of a record is selected by the record variable, followed by the field identifier of the component, separated by a period.

The following lines show examples of record types:

```
TYPE
    CARDTYPE = (NORMAL,WILD);
    COMPLEX =   RECORD
                    R,I : REAL
                END; (* COMPLEX RECORD *)
    DATE =  RECORD
                ORDINAL : 1..31;
                DAY : DAYS;
                MONTH : MONTHS;
                YEAR : 1900..2000
            END; (* DATE RECORD *)
    PLAYINGCARD =   RECORD
                    CASE T : CARDTYPE OF
                        NORMAL : (SUIT : SUITS; RANK : 2..14);
                        WILD : (FACE : (BLANK,JOKER))
                    END; (* PLAYINGCARD RECORD *)
```

If you assume the following declarations,

```
VAR
    S,X,Y : COMPLEX;
    HAND : ARRAY[1..13] OF PLAYINGCARD;
```

then the following are examples of record denotations:

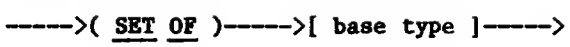
```
S.R := X.R + Y.R; (* THE REAL COMPONENT OF S BECOMES THE SUM OF THE
                     REAL PARTS OF X AND Y *)
HAND[1].T := NORMAL;
HAND[1].SUIT := CLUB;
HAND[1].RANK := 8;
HAND[2].T := WILD;
HAND[2].FACE := JOKER;
```

Note that the tag field is used as any other field is used.

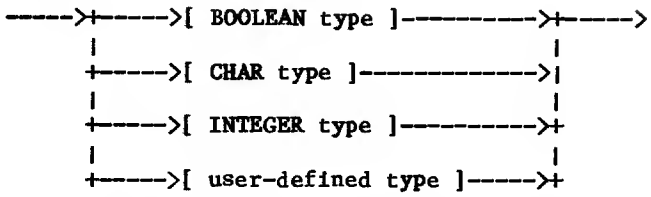
SET Type

A set is a group of subsets of some enumeration type. The type SET describes the group of subsets of some enumeration type.

SET Type

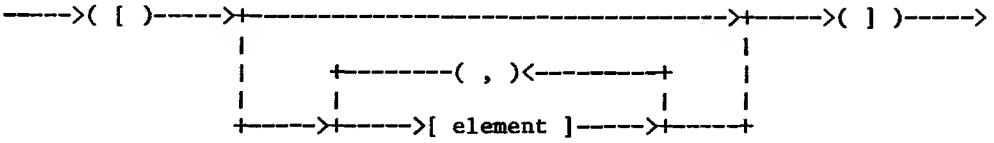


Base Type

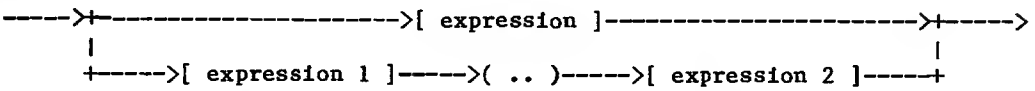


The ordinal number of the largest element cannot exceed 58, and the ordinal number of the smallest element cannot be negative. It follows that a SET type can contain at most 59 elements. SET OF CHAR is illegal.

Set Value



Element



A set value denotes a set consisting of the expression values. The form [M..N] denotes the set of all elements i of the base type such that M <= I <= N. If M > N then [M..N] denotes the empty set. The set expressions must all be of compatible enumeration types. The empty set is denoted [ ] and is compatible with any SET type.

For an example of a SET type, assume that A and B are of type T and T is a SET type. Then the following expression is true:

```
(A - B) + (B - A) = A + B - A * B
```

Set A contains the expressions (A - B) and (A * B). Set B contains the expressions (B - A) and (A * B).

If you assume the following declarations,

```
TYPE
    WORKINGDAYS = SET OF DAYS;
    CHARACTERS = SET OF 'A'..'+';
VAR
    WORKINGDAY : WORKINGDAYS;
    LETTERS, DIGITS, FIRST, FOLLOWING: CHARACTERS;
    LAZY : BOOLEAN;
```

then the following lines are examples of applications of set and set operators:

```
WORKINGDAY := [MONDAY..FRIDAY];
LAZY := NOT(SATURDAY IN WORKINGDAY);
LETTERS := ['A'..'Z'];
DIGITS := ['0'..'9'];
FIRST := LETTERS;
FOLLOWING := FIRST + DIGITS + ['+'];
```

The following relations are all true:

```
FIRST * DIGITS = [ ]
FOLLOWING - (DIGITS + ['+']) = LETTERS
FOLLOWING * FIRST = LETTERS
ORD([MONDAY, TUESDAY, THURSDAY]) = 1 + 2 + 8
```

POINTER DATA TYPE

Pointers are used for constructing dynamic data structures.

Pointer Type

```
----->( ↑ )----->[ type identifier ]----->
```

The type identifier cannot denote a type containing a file type. The type identifier may be defined textually after the pointer type.

The value of a pointer variable is either NIL or a reference to a variable of the specified type. The pointer value NIL belongs to every pointer type; it points to no variable at all. The variable referenced by a pointer is denoted by the pointer variable followed by an arrow (↑).

Referenced Variable

```
----->[ pointer variable ]----->( ↑ )----->
```

Pointer Variable

```
----->[ variable ]----->
```

For example, a list structure can be declared as follows:

```
TYPE
    PLIST = ↑LIST;
    LIST =  RECORD
                INF : ... ;
                NEXT : PLIST
            END; (* LIST RECORD *)
VAR
    HEAD : PLIST;
```

A list structure with two elements can be created as follows:

```
NEW(HEAD);
HEAD↑.INF := ... ;
NEW(HEAD↑.NEXT);
HEAD↑.NEXT↑.INF := ... ;
HEAD↑.NEXT↑.NEXT := NIL;
```

The declaration of a pointer variable causes the computer to allocate space for the pointer. Space is not allocated for a referenced variable until the predefined procedure NEW is called.

The type of a reference variable is the type specified in the declaration of the pointer type.

DATA TYPE COMPATIBILITY

Two operands must be compatible in type. In general,

Two types are compatible if they are the same type.

A subrange type is compatible with the type it is a subrange of.

Two subrange types of the same type are compatible.

Two string types are compatible if they have the same length.

Two set types are compatible if their base types are compatible.

The type of the empty set [ ] is compatible with any set type.

The type of the pointer value NIL is compatible with any pointer type.

The type INTEGER and any subrange type of INTEGER are compatible with the type REAL except in the following cases:

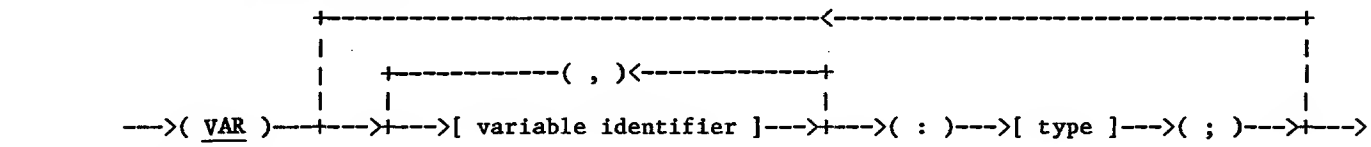
An operand of type REAL cannot be assigned to a variable or function identifier of type INTEGER.

An actual parameter of type REAL cannot be passed to a formal parameter of type INTEGER.

VAR SECTION

The VAR declaration section defines the name and type of a variable. Each variable in the statement part must be declared in the VAR declaration section prior to its use.

Variable Declaration Section



Variable Identifier

You can declare several variables of the same type in a single list of identifiers followed by the type.

If a variable is of array type or record type, a single component is denoted by the identifier followed by a selector specifying that component.

The following is an example of both a TYPE and a VAR declaration section:

```
TYPE
    WEEKEND = FRIDAY..SUNDAY;
VAR
    GOODDAY : WEEKEND;
```

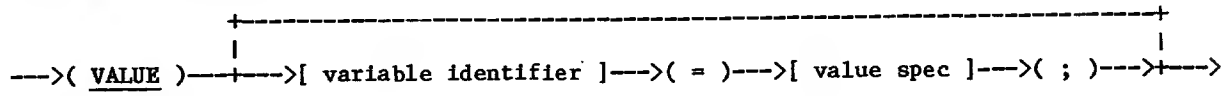
If the variable GOODDAY has the value FRIDAY, then the following relations are true:

```
PRED(GOODDAY) = THURSDAY
SUCC(THURSDAY) = GOODDAY
```

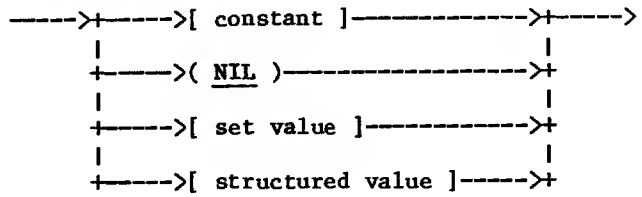
VALUE SECTION (CDC)

The VALUE declaration section initializes the variables that are declared in the statement part.

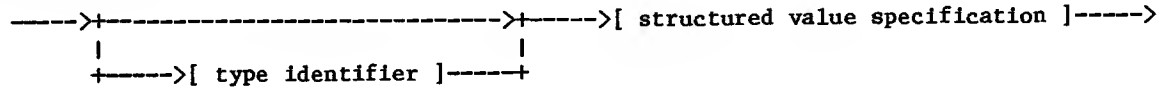
Value Declaration Section



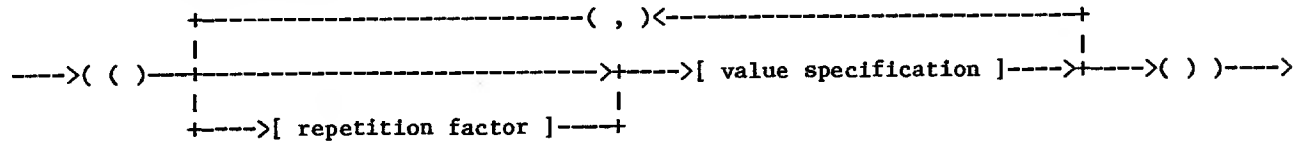
Value Specification



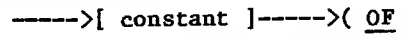
Structured Value



Structured Value Specification



Repetition Factor



A variable of a simple type can be initialized with a constant of the same type.

A variable of pointer type can only be initialized with NIL.

A variable of set type can be initialized with a set value.

A variable of ARRAY or RECORD type can be initialized with a structured value.

A structured value consists of a number of component values, one for each component of the structured type. Each component value must be of the same type as the corresponding component type. If the component type is simple, pointer, or set type, the corresponding component value must follow the rules just given. If a component type is itself an ARRAY or RECORD type, the corresponding component value must be a structured value (this rule is used recursively). A multidimensional array is considered to be an array of arrays.

A type identifier can be present in a structured value. If it is present, it must denote the same type as the type of the variable being initialized.

The type identifier can be omitted, in which case the rules just given apply.

The following is an example of CONST, TYPE, VAR, and VALUE declaration sections:

```
CONST
    N = 5;
    SIZE = 3;
TYPE
    VECTOR = ARRAY[1..N] OF INTEGER;
    NAME = PACKED ARRAY[1..8] OF CHAR;
    NODE = RECORD
               ID : NAME;
               NEXT = ↑NODE
           END;
    MATRIX = ARRAY[1..SIZE, 1..SIZE] OF INTEGER;
    DOUBLEVECTOR = ARRAY[1..2] OF VECTOR;
VAR
    X,Y :VECTOR;
    P,S : NAME;
    M1,M2 : MATRIX;
    D : DOUBLEVECTOR;
    I : INTEGER;
VALUE
    X = VECTOR(1,1,2,2,3);
    P = ('PETER    ');
    S = ('J','O','H','N',4 OF ' ');
    M1 = MATRIX((2,3,5),(7,9,13),(17,19,23));
    M2 = ((3 OF 0),(3 OF 1),(1,2,3));
    D = DOUBLEVECTOR(2 OF VECTOR(N OF 0));
    I = 7;
```

A repetition factor can be used to initialize many array elements with the same value. The constant in a repetition factor must be of type INTEGER.

Packed variables can be initialized with a string literal.

A variable of type CHAR can also be initialized with a string literal.

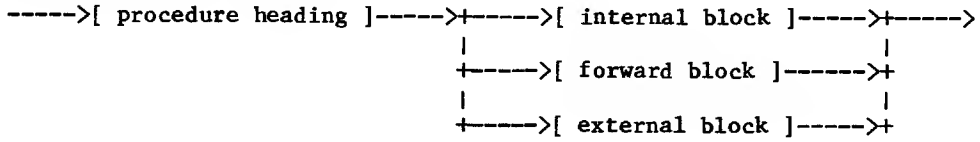
A record with a variant part can be initialized; the tag field value determines which variant is followed. Even if the tag field has no field identifier, the tag field value must be specified to select a variant.

Routine is a general term for a procedure or function. The differences between a procedure and a function are that while both are subprograms, a function returns a result and a function identifier can be used in an expression. This section describes all aspects of routines, including routine declaration, routine parameter lists, directives, and predefined routines.

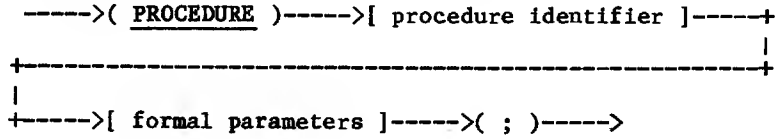
DECLARING A ROUTINE

A procedure is declared in the procedure section in the data declaration and definition part of the block in which it is used.

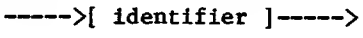
Procedure Declaration Section



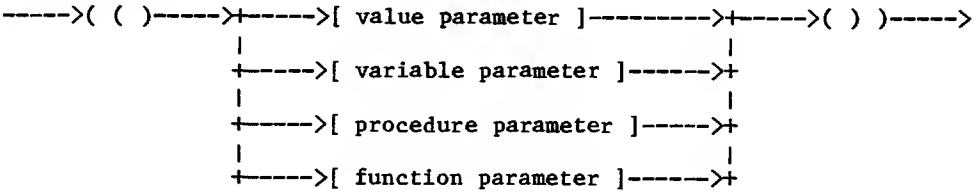
Procedure Heading



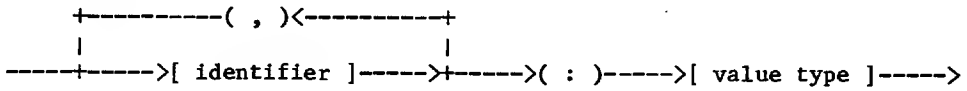
Procedure Identifier



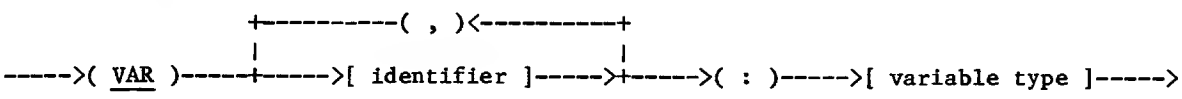
Formal Parameters



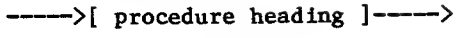
Value Parameter



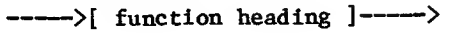
Variable Parameter



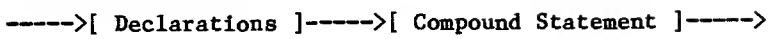
Procedure Parameter



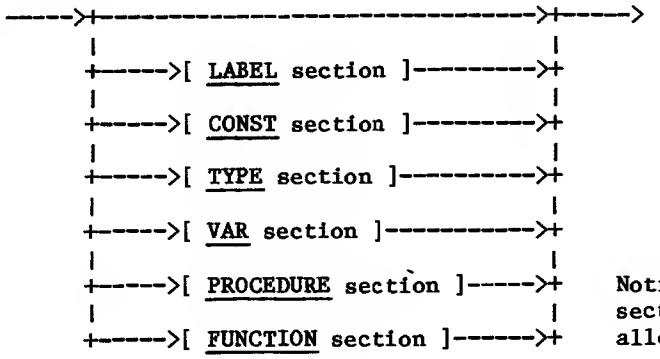
Function Parameter



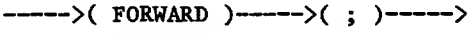
Internal Block



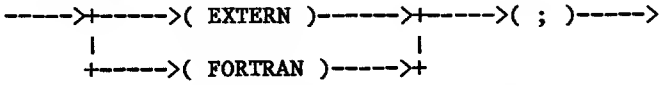
Declarations



Forward Block



External Block



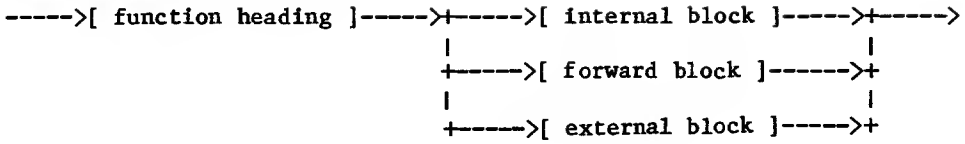
Formal parameters are discussed later in this section under the heading Parameters. The FORWARD, EXTERN (CDC), and FORTRAN (CDC) directives are discussed later in this section under the heading Directives.

The following is an example of a PROCEDURE section:

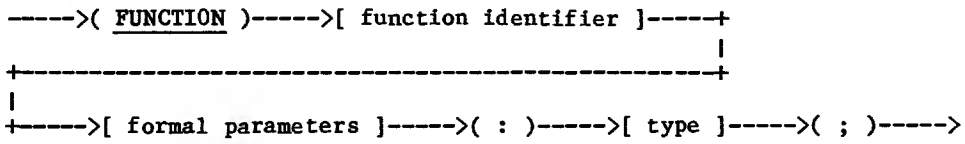
```
PROCEDURE INSERT(ELEMENT : COMPONENTYPE);
.
.
.
PROCEDURE UPDATE(VAR ELEMENT : COMPONENTYPE);
.
.
.
```

A function is declared in the FUNCTION section in the data declaration and definition part of the block in which it is used.

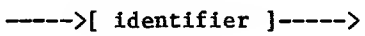
Function Declaration Section



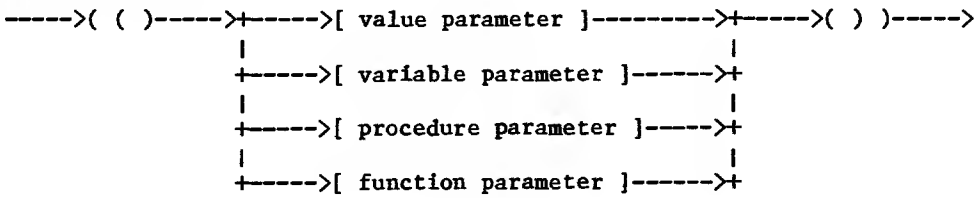
Function Heading



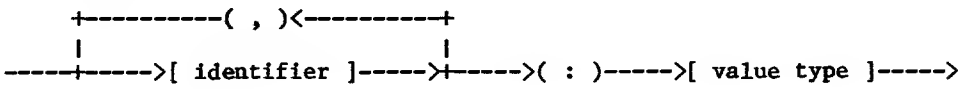
Function Identifier



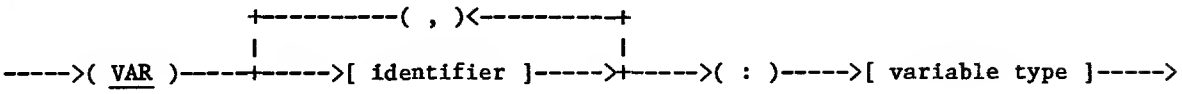
Formal Parameters



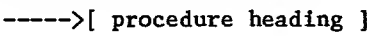
Value Parameter



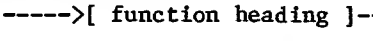
Variable Parameter



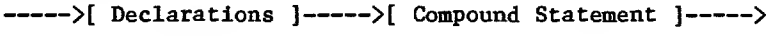
Procedure Parameter



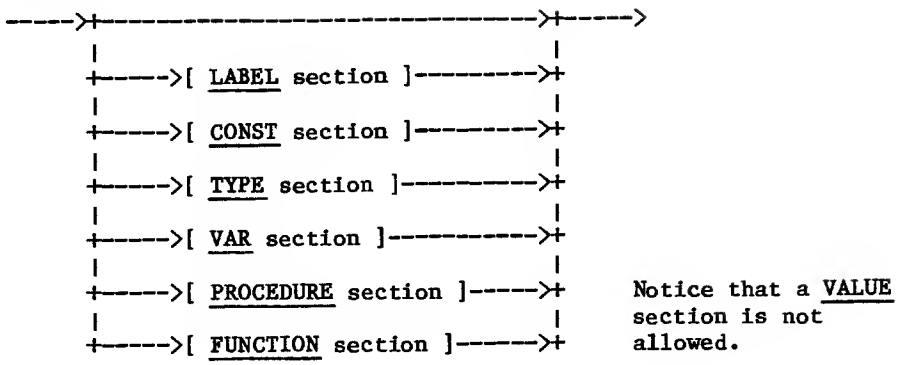
Function Parameter



Internal Block

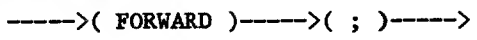


Declarations

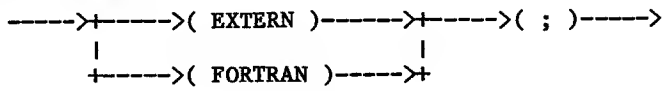


Notice that a VALUE section is not allowed.

Forward Block



External Block



The type identifier specifies the type of the function; the type must be a scalar or pointer type.

The block of a function must contain at least one assignment to the function identifier. If more than one assignment is made, then the last executed assignment is the current value of the function identifier.

Formal parameters are discussed later in this section under the heading Parameters. The FORWARD, EXTERN (CDC), and FORTRAN (CDC) directives are discussed later in this section under the heading Directives.

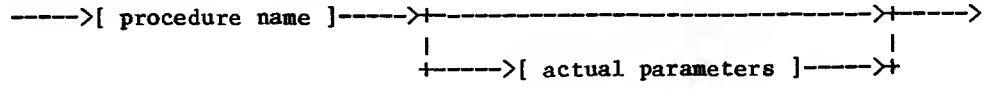
The following is an example of a FUNCTION section:

```
FUNCTION ZERO(LOWER,UPPER : REAL; FUNCTION F(X : REAL) : REAL) : REAL;
.
.
.
FUNCTION MYSQRT(X : REAL) : REAL;
.
.
.
```

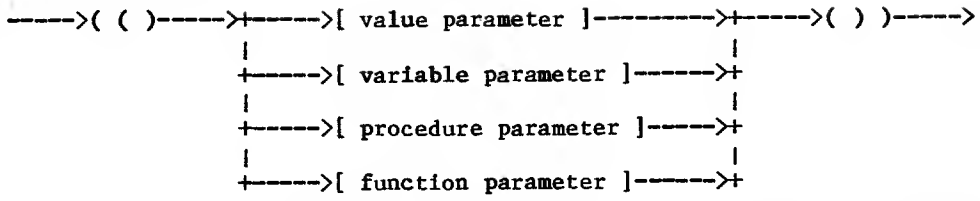
CALLING A ROUTINE

A procedure is called by specifying the procedure name and any actual parameters.

Procedure Call

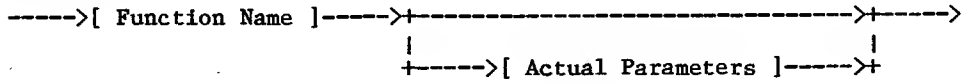


Actual Parameters

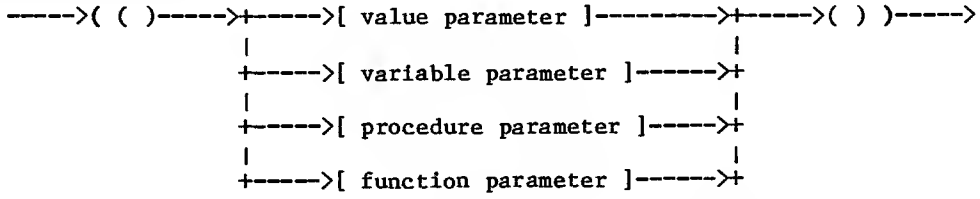


A function is called by specifying the function name and any actual parameters.

Function Call



Actual Parameters



Calling a routine binds the actual parameters to the formal parameters, allocates local variables, and executes the block of statements that define the routine. When the block of statements is completed, local variables are deallocated and execution is resumed with the statement that follows the routine call.

A routine can be called recursively, which means that a routine can call itself within the block that defines it. Routines can be nested to ten levels. At runtime, however, dynamic routine calls can be nested to any level. Variables in a routine are associated with a specific call; they exist from the time the routine is called until the block of statements is done. If a routine is called recursively, several versions of the variables exist simultaneously, one for each uncompleted call.

Actual parameters and binding are discussed in the following paragraphs.

PARAMETERS

A parameter is a variable that is specified on a procedure or function call or on a procedure or function heading. Multiple parameters in a list must be separated by a comma. The collection of parameters, called the argument list, must be enclosed within parentheses.

There are two categories of parameters: formal parameters and actual parameters. Formal parameters appear on a procedure or function call. Actual parameters appear on a procedure or function heading.

Formal and actual parameter pairs can be value parameters, variable parameters, procedure parameters, or function parameters.

Formal and actual parameters are bound, or associated, to each other when the procedure or function is called. How an actual parameters is bound to its formal parameter depends on the kind of parameter pair.

VALUE PARAMETERS

To bind two value parameters (call-by-value), the type of the actual parameter must be compatible with the type of the formal parameter. The actual parameter must be a value. The value of the actual parameter becomes the initial value of the formal parameter. Changes to the formal parameter within the block does not affect the value of the actual parameter.

Call-by-value is preferred when the routine does not return a result. However, call-by-value is not an efficient use of memory when the value parameter is a large structured type.

VARIABLE PARAMETERS

To bind two variable parameters (call-by-reference), the type of the actual and formal parameter must be the same. The actual parameter must be a variable. The value of the actual parameter becomes the initial value of the formal parameter. Changes to the formal parameter within the block can affect the value of the actual parameter.

Each call-by-reference parameter in the function or procedure heading must have the word VAR in front of it in the argument list. The following sequence is an example of a call-by-reference:

```
TYPE
    EXPRESSION = ARRAY[1..72] OF CHAR;
VAR
    STRINGIN, STRINGOUT : EXPRESSION;
PROCEDURE COUNTBLANK(VAR STRIN, STROUT : EXPRESSION);
    .
    .
    .
    BEGIN
        .
        .
        .
        COUNTBLANK(STRIN, STROUT);
        .
        .
        .
    END;
```

A component of a packed structure can be used as a call-by-reference parameter only if it occupies a whole multiple of 60-bit machine words.

An element or a field of a packed variable cannot be used as a call-by-reference parameter; however, the whole packed variable can.

PROCEDURE AND FUNCTION PARAMETERS

To bind two procedure parameters or two function parameters, the actual parameter list must be congruous with the parameter list of the formal parameter. For example, in the following sequence, the procedure PROC1 parameter list must be congruous with the procedure F parameter list or else the statement X(PROC1) will not be valid:

```
PROGRAM MAIN;
PROCEDURE X(PROCEDURE F(A, B : REAL));
    BEGIN
      .
      .
      .
    END; (* X *)
PROCEDURE PROC1(R, S : REAL);
    BEGIN
      .
      .
      .
    END; (* PROC1 *)
BEGIN
    X(PROC1)
END. (* MAIN *)
```

Two formal parameter lists are congruous if they contain the same number of formal parameter sections and if the corresponding formal parameter sections match. Two formal parameter sections match if any of the following is true:

- Both are value parameter specifications, both contain the same number of parameters, and both type identifiers represent the same type.
- Both are variable parameter specifications, both contain the same number of parameters, and both type identifiers represent the same type.
- Both are either procedural parameter specifications or functional parameter specifications, both formal parameter lists are congruous, and both result types (functional parameter specifications only) are the same.
- Both are either value conformant array specifications or variable conformant array specifications, both contain the same number of parameters, and both conformant array schemas are equivalent. Two conformant array schemas are equivalent if all of the following statements are true:

  There is a single index type specification in each conformant array schema.

  The ordinal type identifier in each index type specification denotes the same type.

  Either the component conformant array schemas are equivalent or the type identifiers of the conformant array schemas denote the same type.

  Both conformant array schemas are either packed or not packed.

The following parameter list pairs are not congruous:

```
(A, B : INTEGER)
(A : INTEGER; B : INTEGER)

(A, B : ARRAY[LO..HI : INTEGER] OF REAL)
(A:ARRAY[LO1..HI1:INTEGER] OF REAL; B:ARRAY[LO2..HI2:INTEGER] OF REAL)

(X, Y : REAL; P, Q : CHAR)
(X : REAL; Y : REAL; P, Q : CHAR)
```

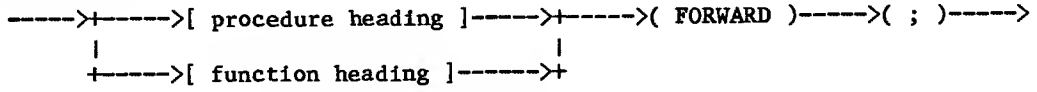
DIRECTIVES

Directives tell the compiler where, in relation to the program unit, the block of the procedure or function occurs. There are three directives: FORWARD, EXTERN (CDC), and FORTRAN (CDC).

FORWARD DIRECTIVE

The FORWARD directive indicates a block that is declared at one point in your program and is defined at a later point in your program. You use a FORWARD directive when two routines call each other; there is a conflict in this situation because the scope rules state that a procedure or function must be defined before it is called. The FORWARD directive postpones the definition of the procedure or function body.

Forward Block



The following is an example of the FORWARD directive:

```
FUNCTION G(X : REAL) : REAL;
FORWARD;
FUNCTION F(X : REAL) : REAL;
.
.
.
BEGIN
    .
    .
    .
    G(X);
    .
    .
    .
END; (* F *)
.
.
.
FUNCTION G;
BEGIN
    .
    .
    .
    F(X);
    .
    .
    .
END; (* G *)
```

EXTERN (CDC) AND FORTRAN (CDC) DIRECTIVES

The EXTERN (CDC) and FORTRAN (CDC) directives indicate an external block. An external block is a block that is declared in your program and is defined outside your program. The EXTERN and FORTRAN directives allow you to access a library of general purpose subprograms. You must use the FORTRAN directive when the subprogram is written in FORTRAN.

External Block

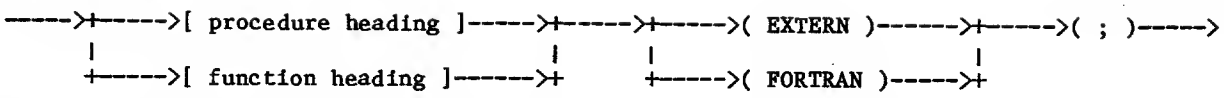


Table 5-1 shows the FORTRAN routine parameter types that correspond to Pascal routine parameter types.

TABLE 5-1. CORRESPONDING FORTRAN AND PASCAL PARAMETER TYPES

Parameter Type in a FORTRAN Routine	Parameter Type in a Pascal Routine	Remarks
INTEGER	INTEGER	With variable parameters of INTEGER, REAL, DOUBLE, and COMPLEX types, a negative zero can be returned by the FORTRAN routine. To eliminate this possibility, you should add a zero to the value upon returning to the Pascal routine.
REAL	REAL	
DOUBLE COMPLEX	RECORD P1 : REAL; P2 : REAL END;	
LOGICAL	INTEGER	Return a negative value for TRUE and a positive value for FALSE.
DIMENSION	ARRAY	You must either transpose multi-dimensional array values before entering a FORTRAN routine or remember that array values are stored by row when manipulating them in the FORTRAN routine. Always set the lower array bound to 1.
SUBROUTINE	PROCEDURE	
FUNCTION	FUNCTION	The result returned to the Pascal routine cannot be COMPLEX, DOUBLE, or a negative zero. To eliminate the possibility of a negative zero, you should add a zero to the value upon returning to the Pascal module.

PREDEFINED PROCEDURES

Predefined procedures are divided into three categories: file handling procedures, dynamic allocation procedures, and transfer procedures. Table 5-2 shows the predefined procedures and gives a brief description of the procedure. See the paragraphs following the table for a more detailed description of the predefined procedures.

TABLE 5-2. PREDEFINED PROCEDURES

Predefined Procedure	Description
DISPOSE(p[,c ...])	Releases the variable that is referenced by p. Any pointer that points to the variable referenced by p becomes undefined and the pointer itself becomes inaccessible.
GET(f)	Advances the pointer in file f to the beginning of the next component and places the value of the component into the buffer variable f↑.
GETSEG(f[,n]) (CDC)	Begins reading at the beginning of the nth segment counting from the current position in file f. GETSEG(f,1) is equivalent to GETSEG(f).
HALT(a) (CDC)	Terminates the program, writes the argument in the dayfile of the job, and produces a dump.
MESSAGE(a) (CDC)	Writes a in the dayfile of the job.
NEW(p[,c ...])	Allocates a new variable and assigns a reference to it.
PACK(a,i,z)	Takes the elements of array a beginning at subscript position i and copies them into packed array z beginning at the first subscript position.
PAGE(f)	Positions the line printer at the top of a new page before printing the next line in file f.
PUT(f)	Appends the value of file buffer variable f↑ to file f.
PUTSEG(f[,n]) (CDC)	Closes the current segment of file f by putting an end-of-segment mark.
READ(f,v[,v ...])	Positions file f and gets the referenced variables. If a file is not specified, INPUT is assumed.

(Continued on next page)

TABLE 5-2. PREDEFINED PROCEDURES

Predefined Procedure	Description
READLN(f,v[,v ...])	Gets the referenced variables from file f. When a READLN is completed, any remaining values on the current input line, including an end-of-line, are discarded. The first The first value on the next line in file f will be read next.
RESET(f)	Positions file f to the beginning-of-information. A RESET(f) must be done on every input file except INPUT.
REWRITE(f[,n]) (CDC)	Empties file f and allows it to be written to. A REWRITE(f) must be done on every output file except OUTPUT.
UNPACK(z,a,i)	Takes the elements beginning at the first subscript position of packed array z and copies them into array a beginning at subscript position i.
WRITE(f,v[,v ...])	Transforms the expressions into a sequence of characters and puts the sequence onto file f. If a file is not specified, OUTPUT is assumed.
WRITELN(f,v[,v ...])	Terminates the current line in file f by putting an end-of-line mark. If a file is not specified, OUTPUT is assumed.

DISPOSE(p[,c ...])

Releases the variable referenced by p. The case constants c1 through cn must be listed in increasing order as they appear within the variant parts.

If the associated type contains variants and NEW(p,c1, ..., cn) is used to allocate the variable, then n must be equal to the number of case constants in the variant part.

If the associated type contains variants and NEW(p,c1, ..., cn) is used to allocate the variable, then DISPOSE(p,c1, ..., cn) must be used to release the variable.

GET(f)

Advances the pointer in file f to the beginning of the next component. F can be a file, textfile, segmented file, or segmented textfile. The value of the file buffer becomes the content of this component. If no next component exists, EOF(f) becomes true and the value of f↑ is undefined. If EOF(f) is true prior to the execution of GET(f), then the call results in the runtime error message: TRIED TO READ PAST EOS/EOF.

GETSEG(f[,n]) (CDC)

Advances the pointer in file f to the beginning of the next segment. F can be a segmented file or segmented textfile. The file buffer f↑ becomes the first component of the next segment. If no next segment is present, then execution is terminated and the runtime error message: TRIED TO READ PAST EOS/EOF is given. GETSEG can be applied only to a segmented file that is being read from.

GETSEG(f,n) advances the pointer in file f to the beginning of the nth segment counting from the current position of the file. The file buffer f↑ becomes the first component of the nth segment. N > 0 implies counting segments in the forward direction. N = 0 means the current segment. If no nth segment (n >= 0) is present, then EOF(f) becomes true and f↑ becomes undefined. N < 0 implies counting segments in the reverse direction. If the file is positioned at segment number m, where m < -n, then GETSEG(f,n) is equivalent to RESET(f).

HALT(a) (CDC)

Closes external files, terminates the program with a CPU abort, places a in the dayfile of the job, and produces a dump. Argument a must be a string.

MESSAGE(a) (CDC)

Places a in the dayfile of the job. Argument a can be CHAR or string type.

NEW(p[,c ...])

Allocates a new variable of the same type as the argument and assigns a reference to the argument.

In the case where the type associated with p is a record type and the field has variants, the form NEW(p,c1, ..., cn) can be used. C1, ..., cn is a list of constant selectors used to determine the size of the allocated variable. The size is as if the variable was declared a record type with the field list formed by the following rule of selection: first, the variant corresponding to the selector c1 is selected, then, the field list of this variant is formed by using the selectors c2, ..., cn (by a recursive application of this rule), finally, the so-far-formed field list is prefixed by the tag field (if nonempty) and is substituted for the variant part.

The above description does not imply any assignment to the tag fields.

The variant of the allocated variable must not be changed, and assignment to the entire variable is not allowed. However, the value of single components can be altered.

If you assume the following declarations:

```
CONST
    MAXVAL = 50;
TYPE
    PATOM = ↑ATOM;
    ATOM =  RECORD
                NAME : ALFA;
                NUMBER : INTEGER;
                WEIGHT : REAL;
                OCCUPIED : SET OF 1.. MAXVAL;
                BINDINGS : ARRAY[1..MAXVAL] OF PATOM;
                CHARGE : (PLUS,MINUS,NEUTRAL);
                SATURATED : BOOLEAN
            END; (* RECORD *)
VAR
    A : ATOM;
```

then the following statements give all the names of the atoms to which A is bound:

```
WITH A DO
    FOR I := 1 TO MAXVAL DO
        IF I IN OCCUPIED THEN
            WRITELN(I,BINDINDS[I]↑.NAME);
```

If you assume the following declarations:

```
VAR
    P : ↑PLAYINGCARD;
```

then NEW(P,WILD) allocates a variable whose size is as if the variable had been of the type Q defined as

```
TYPE
    Q = RECORD
            T : CARDTYPE;
            FACE : (BLANK,JOKER)
        END; (* RECORD *)
```

PACK(a,i,z)

Takes the elements of array a beginning at subscript position i and copies them into packed array z beginning at the first subscript position. Assume that a and p are variables of the following types:

```
A: ARRAY[M..N] OF T;
P: PACKED ARRAY[U..V] OF T;
```

When (ORD(N) - ORD(I)) >= (ORD(V) - ORD(U)) where M <= I and the index types of the arrays A and P and the type of I are compatible, then PACK(A,I,P) is equivalent to:

```
K := I;
FOR J := U TO V DO BEGIN
    P[J] := A[K];
    K := SUCC(K)
END; (* FOR *)
```

PAGE(f)

Positions the lineprinter. Argument f must be a textfile. PAGE(f) is equivalent to the following sequence:

```
WRITELN(f);
WRITE(f,'1');
```

The '1' forces the lineprinter to the top of a new page.

PUT(f)

Appends the value of the buffer variable f↑ to the file f. F can be a file, textfile, segmented file, or segmented textfile. The value of f↑ becomes undefined. If the value of EOF(f) or EOS(f) is false prior to the execution of the PUT(f), then the call results in the runtime error message: TRIED TO WRITE WHILE NOT EOS/EOF. Otherwise the value of EOF(f) remains true.

PUTSEG(f[,n]) (CDC)

Closes the current segment (an end-of-segment mark is written onto f). F can be a segmented file or segmented textfile. PUTSEG(f) is only allowed if EOF(f) is true. PUTSEG can be applied only to a segmented file that is being written to.

READ(f,v[,v ...])

READ(f,v) reads a sequence of characters from the file f through the file buffer f↑ using GET(f). F can be a textfile or a segmented textfile. If f is omitted, then the predefined file INPUT is assumed. The first significant character is the character in f↑.

READ(f,v) is equivalent to the following sequence:

```
v := f↑;
GET(f);
```

READ(f,v1, ..., vn) is a shorthand notation for the following sequence:

```
BEGIN
    READ(f,v1);
    READ(f,v2);
    .
    .
    .
    READ(f,vn)
END;
```

V must be of a type compatible with the type of the components in the file f.

If v is INTEGER, a sequence of digits is transformed into a (decimal) value and then assigned to v. Leading blanks and leading end-of-line marks are skipped. The character sequence that follows must be consistent with the syntax for decimal integers given in section 2. If not, execution is terminated and a runtime error message is given. Trailing blanks are skipped (if the file buffer f↑ is left at the first nonblank character after the number or is left at the end-of-line mark).

If v is REAL, a sequence of characters is transformed into a real value and then assigned to v. Leading blanks and leading end-of-line marks are skipped. The character sequence that follows must be consistent with the syntax for real numbers given in section 2. If not, execution is terminated and a runtime error message is given. Trailing blanks are skipped (if the file buffer is left at the first nonblank character after the real number or is left at the end-of-line mark).

READLN(f,v[,v ...])

Skips to the beginning of the next line of f. F can be a textfile or segmented textfile. If f is omitted, then the predefined file INPUT is assumed. Subsequently, f↑ becomes the first character of the next line, if any. READLN(f,v) is equivalent to the following statement:

```
WHILE NOT EOLN(f) DO
    GET(f);
```

READLN(f,v) is also equivalent to the following sequence:

```
BEGIN
    READ(f,v);
    READLN(f)
END;
```

READLN(f,v1, ..., vn) is equivalent to the following sequence:

```
BEGIN
    READ(f,v1, ..., vn);
    READLN(f);
END;
```

RESET(f)

Positions file f to the beginning-of-information; the file can now be read. F can be a file, textfile, segmented file, or segmented textfile. The file buffer f↑ contains the first component of the file. If file f is empty, then the value of f↑ is undefined and EOF(f) is TRUE. RESET(f) must be specified for all files, except INPUT, before a READ or READLN operation on the file.

REWRITE(f[,n]) (CDC)

REWRITE(f) positions file f to the beginning-of-information; the file can now be written to. F can be a file, textfile, segmented file, or segmented textfile. If file f is empty, then the value of f↑ is undefined and EOF(f) is TRUE. REWRITE(f) must be specified for all files, except OUTPUT, before a write operation is performed on the file.

REWRITE(f,n) positions file f to the beginning of the nth segment counting from the current position. The current segment number is not accessible after execution of the REWRITE(f,n) statement. If file f is empty, then the value of f↑ is undefined and EOS(f) is TRUE.

N > 0 implies counting segments in the forward direction. N = 0 implies the current segment. N < 0 implies counting segments in the reverse direction.

If file f is positioned to n, where n >= 0 but is not valid for file f, then file f is positioned to the end of the last segment and EOF(f) is TRUE.

If file f is positioned to m, where m < -n, then REWRITE(f,m) is equivalent to REWRITE(f).

UNPACK(z,a,i)

Takes the elements from the first subscript position of packed array z and copies them into array a beginning at subscript position i. Assume that A and P are variables of the following types:

```
A: ARRAY[M..N] OF T;
P: PACKED ARRAY[U..V] OF T;
```

When (ORD(N) - ORD(I)) >= (ORD(V) - ORD(U)); M <= I; and the index types of the arrays A and P and the type of I are compatible, then UNPACK(A,I,P) is equivalent to:

```
K := I;
FOR J := U TO V DO BEGIN
    A[K] := P[J];
    K := SUCC(K)
END; (* FOR *)
```

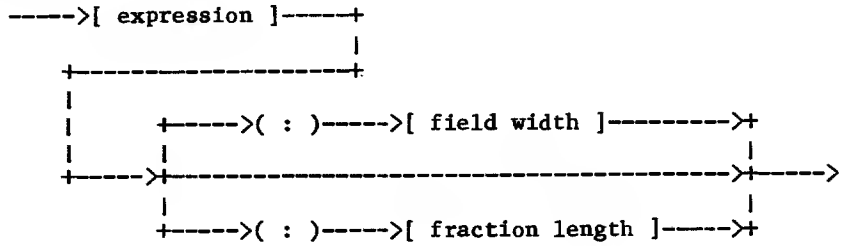
Where J denotes an auxiliary variable that is not used elsewhere in the program.

WRITE(f,v[,v ...])

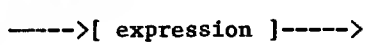
Transforms the expressions into a sequence of characters and puts the sequnce onto file f. F can be a textfile or a segmented textfile. If f is omitted, then the predefined file OUTPUT is assumed.

The parameters in the predefined procedures WRITE and WRITELN must have the following form:

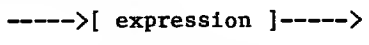
Parameter



Field Width



Fraction Length



The first expression, which is the value to be written, can be INTEGER, BOOLEAN, CHAR, REAL, or STRING type. The fraction length can be given only when the expression is REAL type. The field width indicates the minimum number of characters to be written. If the field width is longer than needed, the value is written right-justified. The field width must be an integer expression with a value greater than or equal to 0. If the field width is omitted, a default value is chosen in accordance with table 5-3.

TABLE 5-3. DEFAULT FIELD WIDTHS

Type	Default Field Width	Remarks
INTEGER	10	If the field width is too short, the necessary number of additional character positions are used.
BOOLEAN	10	If the field width is 5 or more either of the strings ' TRUE' or 'FALSE' is written. If the field width is 0, 1, 2, 3, or 4 either of the characters 'T' or 'F' is written.
CHAR	1	If the field width is 0, the default field width 1 is used.
REAL	22	If fraction length is not specified the value will be written with 1 digit before the decimal point; 13 digits after the decimal point; and a scaling exponent written as E±ddd (floating point notation). If fraction length is specified, the fraction length must be at least two less than the field width. The fraction length specifies the number of digits to follow the decimal point. If the fraction length is specified no exponent is written (fixed point notation). If the field width is too short the necessary number of additional character positions is used.
string	length of string	If a nonzero field width less than the length of the string is specified, the right part of the string is truncated. If a field width equal to 0 is specified the entire string is written.

WRITE(f,v) is equivalent to the following sequence:

```
f↑ := v;
PUT(f);
```

V must be of a type compatible with the type of the components in the file v.

WRITE(f,v1, ..., vn) is equivalent to the following sequence:

```
BEGIN
    WRITE(f,v1);
    WRITE(f,v2);
    .
    .
    .
    WRITE(f,vn)
END;
```

WRITELN(f,v[,v ...])

Terminates the current line of f and writes an end-of-line mark. F can be a textfile or a segmented textfile. The WRITELN statement may append some extra blanks to the line due to peculiarities in the representation of end-of-line mark in the NOS operating system.

WRITELN(f,v1, ..., vn) is equivalent to the following sequence:

```
BEGIN
    WRITE(f,v1, ..., vn);
    WRITELN(f)
END;
```

PREDEFINED FUNCTIONS

Predefined functions are divided into four categories: arithmetic functions, transfer functions, ordinal functions, and Boolean functions. Table 5-4 shows the predefined functions and gives a brief description of the function. See the paragraphs following the table for a more detailed description of the predefined functions.

TABLE 5-4. PREDEFINED FUNCTIONS

Predefined Function	Argument Type	Result Type	Description
ABS(a)	INTEGER REAL	INTEGER REAL	Returns the absolute value of a.
ARCTAN(a)	INTEGER REAL	REAL REAL	Returns the arctangent of a.
CARD(a) (CDC)	SET	INTEGER	Returns the cardinality of a.
CHR(a)	INTEGER	CHAR	Returns the character that has ordinal number a.
CLOCK (CDC)	None	INTEGER	Returns the current used CPU-time in milliseconds.
COS(a)	INTEGER REAL	REAL REAL	Returns the cosine of a.
DATE(a) (CDC)	ALFA	ALFA	Assigns the current date to a.
EOF(f)	FILE	BOOLEAN	Returns a TRUE value if an end-of-file mark has been reached on file f and a FALSE value if an end-of-file mark has not been reached.
EOLN(f)	FILE	BOOLEAN	Returns a TRUE value if an end-of-line mark has been reached on file f and a FALSE value if an end-of-line mark has not been reached.

(Continued on next page)

TABLE 5-4. PREDEFINED FUNCTIONS

(Continued)

Predefined Function	Argument Type	Result Type	Description
EOS(f) (CDC)	FILE	BOOLEAN	Returns a TRUE value if an end-of-segment mark has been reached on file f and a FALSE value if an end-of-segmented mark has not been reached.
EXP(a)	INTEGER REAL	REAL REAL	Returns the value of $E^{(a)}$.
EXPO(a) (CDC)	INTEGER REAL	REAL REAL	Returns the value of $E^{(a)}$ in binary representation.
LN(a)	INTEGER REAL	REAL REAL	Returns the value of the natural logarithm of a.
ODD(a)	INTEGER	BOOLEAN	Returns a TRUE value if a is odd and a FALSE value if a is even.
ORD(a)	BOOLEAN CHAR POINTER SET	INTEGER	Returns the position of a in the set of values defined by the type of a.
PRED(a)	Scalar	Scalar	Returns the predecessor of a. A cannot be REAL. If a does not exist an error will occur.
ROUND(a)	REAL	INTEGER	Returns a rounded to the nearest integer.
SIN(a)	INTEGER REAL	REAL REAL	Returns the sine of a.
SQR(a)	INTEGER REAL	INTEGER REAL	Returns the square of a.
SQRT(a)	INTEGER REAL	REAL REAL	Returns the square root of a.
SUCC(a)	SET	SET	Returns the successor of a.
TIME(a) (CDC)	ALFA	ALFA	Assigns the current time to a.
TRUNC (a[,n]) (CDC)	REAL	INTEGER	Returns either the largest integer $\leq$ a if a $\geq$ 0 or the smallest integer $\geq$ a if a < 0.
UNDEFINED (a) (CDC)	REAL	BOOLEAN	Returns a TRUE value if a is out of range or indefinite or a FALSE value if a is not out of range or indefinite.

ABS(a)

Returns the absolute value of a. Argument a can be either INTEGER or REAL type; the result type is the same as the argument type.

ARCTAN(a)

Returns the arctangent of a in radians. Argument a can be either INTEGER or REAL type; the result is always REAL type.

CARD(a) (CDC)

Returns the cardinality of a. Cardinality is the number of elements in the set. Argument a must be of SET type; the result is always INTEGER type.

CHR(a)

Returns the character that has ordinal position a in the character set used at your installation. CHR(a) is only defined in the range [0..63]. Argument a must be INTEGER type; the result is CHAR type.

CLOCK (CDC)

Returns the current used CPU-time in milliseconds. The result is INTEGER type.

COS(a)

Returns the cosine of a. Argument a can be either INTEGER or REAL type; the result is always REAL type.

DATE(a) (CDC)

Assigns the current date to a in the form: YY/MM/DD (year/month/day). Argument a must be ALFA type.

EOF(f)

Returns a TRUE value if the end-of-file mark has been reached and a FALSE value if the end-of-file mark has not been reached. Argument f is FILE type; the result is BOOLEAN type. EOF(f) always implies EOS(f).

EOLN(f)

Returns a TRUE value if the end-of-line mark has been reached and a FALSE value if the end-of-line mark has not been reached. Argument f is FILE type; the result is BOOLEAN type.

EOS(f) (CDC)

Returns a TRUE value if an end-of-segment mark has been reached and a FALSE value if an end-of-segment mark has not been reached. Argument f is FILE type; the result is BOOLEAN type. EOS(f) can be applied only to a segmented file.

You can use the predefined procedure GET(f) only when EOS(f) is FALSE; you can use the predefined procedures PUT(f) and PUTSEG(f) only when EOS(f) is TRUE.

EXP(a)

Returns the value of $E^{(a)}$. Argument a can be either INTEGER or REAL type; the result is always REAL type.

EXPO(a) (CDC)

Returns the value of $E^{(a)}$ in binary representation. Argument a must be REAL type; the result is INTEGER type.

LN(a)

Returns the natural logarithm of a. Argument a can be either INTEGER or REAL type; the result is always REAL type.

ODD(a)

Returns a TRUE value if a is odd and a FALSE value if a is even. Argument a must be INTEGER type; the result is BOOLEAN type.

ORD(a)

Returns the ordinal, or position, of a in the set of values defined by the type of a. ORD(pointer value) is the integer representation of the pointer. ORD(Boolean value) is zero if the value is FALSE and one if the value is TRUE. Ord(Subrange value), given the following declaration:

```
VAR
    A : INTEGER;
    B : MIN..MAX;
```

and the statement A = B, is equivalent to ORD(A) = ORD(B).

PRED(a)

Returns the predecessor of a in the set of values defined by the type of a. Argument a can be any scalar type except REAL type; the result type is the same as the argument type. If the argument is the first (smallest) value of the type, then the result may be undefined.

ROUND(a)

Returns a rounded (not truncated) to the nearest integer. If $a \geq 0$, then ROUND(a) = TRUNC(a + 0.5). If $a < 0$, then ROUND(a) = TRUNC(a - 0.5). Argument a must be REAL type; the result is INTEGER type.

The difference between ROUND and TRUNC is illustrated by the following examples:

```
TRUNC(1.6) = 1      ROUND(1.6) = 2
TRUNC(-1.6) = -1    ROUND(-1.6) = -2
TRUNC(2.4) = 2      ROUND(2.4) = 2
```

The operators equal to (=) and not equal to (<>) should be used with great care on real arguments because of round-off errors that often result from the representation of real values, as in the following examples:

```
(1.00000 - 0.00001) = 0.99999      FALSE
SQR(SQRT(2)) = 2                   FALSE
(4.0 * 0.25) = 1                   TRUE
(10000 * 0.0003) = 3               TRUE
(1000000 * 0.000003) = 3           FALSE
```

SIN(a)

Returns the sine of a. Argument a can be either INTEGER or REAL type; the result is always REAL type.

SQR(a)

Returns the square of a. Argument a can be either INTEGER or REAL type; the result type is the same as the argument type.

SQRT(a)

Returns the square root of a. Argument a can be either INTEGER or REAL type; the result is always REAL type.

SUCC(a)

Returns the successor of a in the set of values defined by the type of a. Argument a can be any scalar type except REAL type; the result type is the same as the argument type. If the argument is the last (greatest) value of the type the result may be undefined.

TIME(a) (CDC)

Assigns the curent time to a in the form: HH.MM.SS. (hour.minute.seconds.). Argument a must be ALFA type.

TRUNC(a[,n]) (CDC)

Returns a as an integer with the same sign as argument a. Argument a must be REAL type; the result is INTEGER type.

TRUNC can also be applied to two arguments; the first argument must be REAL, the second argument must be INTEGER. TRUNC(a,n) is equal to TRUNC(a * 2**n).

The difference between ROUND and TRUNC is illustrated in the following examples:

```
TRUNC(1.6) = 1      ROUND(1.6) = 2
TRUNC(-1.6) = -1    ROUND(-1.6) = -2
TRUNC(2.4) = 2      ROUND(2.4) = 2
```

UNDEFINED(a) (CDC)

Returns a TRUE value if a is out of range or indefinite and a FALSE value if a is not out of range or indefinite. Argument a must be REAL type; the result is BOOLEAN type.

STATEMENTS 6

A Pascal program must contain a program heading part, a declaration and definition part, and a statements block. This section describes Pascal statements.

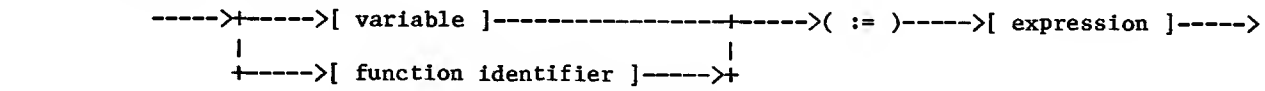
Statements manipulate the defined and declared data items. A collection of statements can be grouped into a compound statement by enclosing them within a BEGIN and an END statement.

The statement(s) in the statement part are executed sequentially in the same order as they appear.

ASSIGNMENT STATEMENT

The assignment statement replaces the current value of a variable or function identifier with the value of an expression.

Assignment Statement



The following is an example of the assignment statement:

```
BLANK := ' ';
```

The variable or function identifier and expression must be of compatible types. An assignment can be made to a variable of any type except file type; an assignment cannot be made to the file buffer of a file.

At least one assignment must be made to a function identifier within its block of statements. An assignment can be made to a function identifier within a procedure or function that is nested within the function that is being defined. For example:

```
FUNCTION OUTER (J : INTEGER) : INTEGER;
    PROCEDURE INNER (K : INTEGER);
    BEGIN
        IF K >= 10 THEN OUTER := K
        ELSE CUTER := 10
    END; (* INNER *)
BEGIN
    INNER (J)
END; (* OUTER *)
```

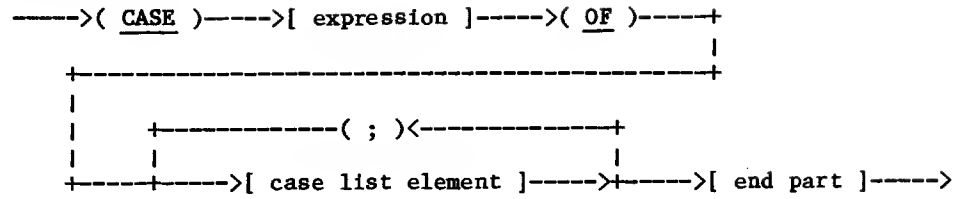
If a function identifier is redefined in an inner scope of the function definition and an assignment is made to the function identifier, then the assignment requirement is not fulfilled. A value must be assigned to the function identifier in the scope of the function identifier.

The value returned by a function is the last value that was assigned to it.

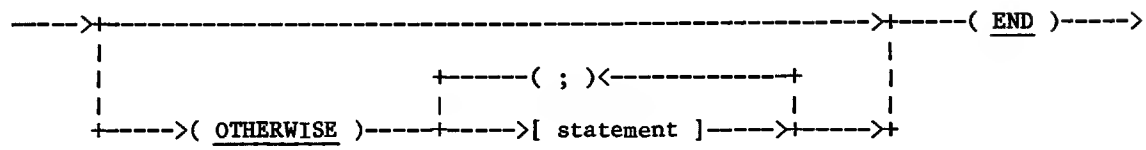
CASE STATEMENT

The CASE statement describes multiple paths of execution; the selection of a path depends upon the value of an expression. OTHERWISE is a CDC extension.

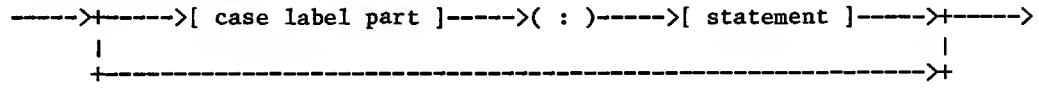
CASE Statement



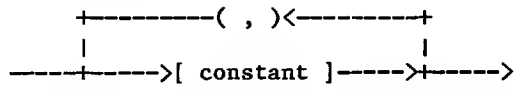
End Part



Case List Element



Case Label Part



The following are examples of the CASE statement:

```
VAR
    MONTH : MONTHS;
    SUIT : SUITS;
    SEASON : SEASONS;
    COLOR : COLORS;
.
.
.
CASE MONTH OF
    DECEMBER, JANUARY, FEBRUARY : SEASON := WINTER;
    MARCH, APRIL, MAY : SEASON := SPRING;
    JUNE, JULY, AUGUST : SEASON := SUMMER;
    SEPTEMBER, OCTOBER, NOVEMBER : SEASON := AUTUMN
END; (* CASE MONTH *)
CASE SUIT OF
    CLUB, SPADE : COLOR := BLACK
OTHERWISE
    COLOR := RED
END; (* CASE SUIT *)
```

The expression must be of enumeration type.

A CASE list element is a statement labeled by one or more constants. These constants must all be the same enumeration type as the expression and must be distinct.

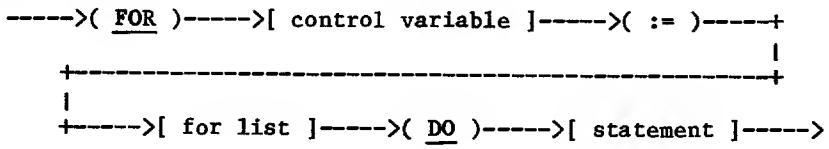
The CASE statement is translated into a jump table that is limited in size. Therefore, all labels, such as L1 and L2, must be chosen so that ABS(ORD(L1) - ORD(L2)) > 1000.

The statement labeled by the current value of the expression is selected for execution. If no such label is present, the statements following OTHERWISE are selected for execution. If OTHERWISE is not included in the CASE statement and the T+ compiler option (see section 7) is specified, then the runtime error message INDEX OR CASE EXPR OUT OF RANGE is given and program execution terminated. If OTHERWISE is not included in the CASE statement and the T- compiler option is specified, then no statement is selected for execution. When the selected statement has been executed, then the CASE statement is done.

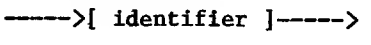
FOR STATEMENT

The FOR statement isolates a group of statements that is to be executed a specified number of times.

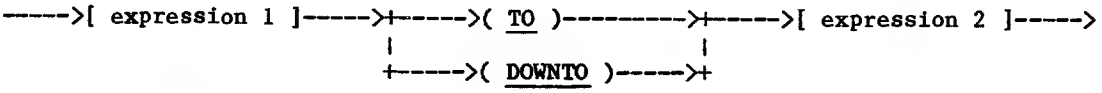
FOR Statement



Control Variable



For List



The following is an example of the FOR statement:

```
FOR CHARACTER := 'A' TO ';' DO
    WRITE (ORD (CHARACTER) : 3);
```

The control variable must be a local variable of enumeration type; the control variable cannot be a global variable or a formal parameter. The control variable can either be incremented (in steps of 1) from expression 1 TO expression 2, or decremented (in steps of 1) from expression 1 DOWNTO expression 2. The value of the control variable is undefined after the FOR statement is done. For example, the value of I = J in the following sequence is undefined:

```
FOR I := 1 TO N DO BEGIN
.
.
.
END; (* FOR *)
IF I = J THEN
```

A control variable cannot be assigned a value in a statement that is inside the FOR statement. For example, the following statement is invalid:

```
FOR I := 1 TO 100 DO I := A + SQR(A);
```

A control variable cannot be used as a control variable in a nested FOR statement. For example, the following sequence is invalid:

```
FOR I := 1 TO 100 DO
    FOR I := 2 TO 100 DO
```

A control variable cannot be an actual parameter in a function or procedure call within a FOR statement. For example, the following sequence is invalid:

```
FOR I := 1 TO 100 DO GETSTUDENTS (I : INTEGER);
```

A control variable cannot be a parameter in a READ or READLN statement.

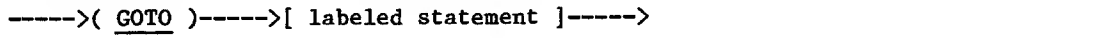
The for list expresses the size of the interval and the order of progression. The FOR list must contain two expressions that are the same enumeration type; both expression types must be compatible with the type of the control variable.

The expressions are evaluated only at the time the FOR statement is compiled. If expression 1 is greater than expression 2 and the increment operation (TO) is specified, then the statement is not executed. If expression 1 is less than expression 2 and the decrement operation (DOWNTO) is specified, then the statement is not executed. Either expression can contain a variable.

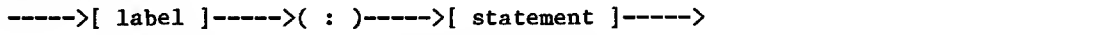
GOTO STATEMENT

A GOTO statement is a means of transferring control to an arbitrary place in a program block. GOTO statements are not a prefered programming construct; any GOTO statement can be constructed using WHILE and IF statements and auxiliary Boolean variables.

GOTO Statement



Labeled Statement



The following is an example of the GOTO statement:

```
    LABEL
        10;
    .
    .
    .
    GOTO 10;
    .
    .
    .
10 : GETSCORE;
```

A GOTO statement can contain a label if at least one of the following conditions is true:

- The label prefixes a statement that contains the GOTO statement, as in the following sequence:

```
1 : IF (A < B) THEN GETSTUDENTS
    ELSE BEGIN
        GETSCORES;
        GOTO 1
    END; (* ELSE *)
```

- The label prefixes a statement that is one of a sequence of statements and the GOTO statement is contained in another statement that is one of the same sequence of statements, as in the following sequence:

```
REPEAT
    1 : GETSTUDENTS;
    IF (COUNT < 100) THEN BEGIN
        GETSCORES;
        GOTO 1
    END; (* IF *)
    REPORT;
UNTIL DONE;
```

- The label prefixes an unnested statement in the block of a function or procedure and the GOTO statement occurs in another function or procedure that is nested within the first function or procedure, as in the following sequence:

```
PROCEDURE OUTER;
LABEL
    1;
PROCEDURE INNER;
BEGIN
    IF (COUNT < 100) THEN GOTO 1
END; (* INNER *)
BEGIN
    INNER;
    IF (A < B) THEN BEGIN
    .
    .       <---------------------- The label cannot prefix a statement inside the
    .                               IF statement because the statements are nested
    END; (* IF *)
    1 : FINALIZE
END; (* OUTER *)
```

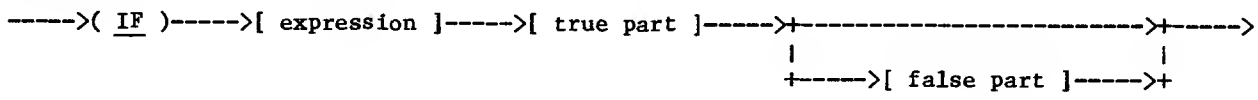
All labels must be declared in the LABEL section of the program block in which it is used. The statement after the colon cannot be a labeled statement.

If nested routines use the same label, the innermost label will be effective. The result of a jump to a statement within an IF, WHILE, REPEAT, WITH, FOR, or CASE construct is undefined.

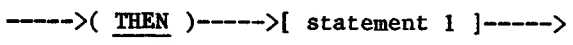
IF STATEMENT

The IF statement defines paths that can be taken during program execution. The path that is taken depends upon the result of the Boolean expression contained in the statement.

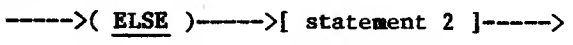
IF Statement



True Part



False Part



Statement 1 will be executed if the value of the expression is true. The statement following statement 1, in this case statement 2, will be executed if the value of the expression is false.

The following are examples of the IF statement:

```
IF X > Y THEN BEGIN
    MIN := Y;
    MAX := X
END; (* IF X>Y *)
ELSE BEGIN
    MIN := X;
    MAX := Y
END; (* ELSE *)
```

An IF statement can reduce the number of statements in your program, and thereby clarify the program. For example, the following CASE statement:

```
CASE B OF
    TRUE : S1;
    FALSE : S2
END;
```

is equivalent to the IF statement:

```
IF B THEN S1 ELSE S2;
```

However, an IF statement can complicate your program. For example, the following IF statement:

```
IF A = B THEN FOUND := TRUE
ELSE FOUND := FALSE;
```

is equivalent to the assignment statement:

```
FOUND := A = B;
```

Ambiguity can arise from some constructions of the IF statement. Some ambiguous IF statements can be resolved by inserting a BEGIN and an END statement around the group of statements that you want to be executed sequentially. For example, the following ambiguous IF statement:

```
IF E1 THEN IF E2 THEN S1 ELSE S2
```

can be clarified by rewriting it as follows:

```
IF E1 THEN BEGIN
    IF E2 THEN S1
    ELSE S2
END; (* IF *)
```

A more subtle form of ambiguity can arise from when an expression is evaluated. For example, the following IF statements yield the same result, but are evaluated differently:

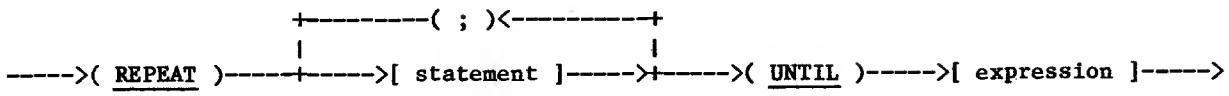
```
IF (I <= N) AND (TABLE[I] = KEY) THEN S;

IF I <= N THEN IF TABLE[I] = KEY THEN S;
```

In the case where I > N, the first statement will evaluate TABLE[I] = KEY and probably cause an index error.

REPEAT STATEMENT

The REPEAT statement specifies that a sequence of statements is to be executed one or more times.

REPEAT Statement



The following is an example of the REPEAT statement:

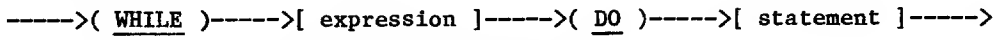
```
REPEAT
    READ(LINE);
    INSERTDELIM;
    COUNT := COUNT + 1
UNTIL COUNT = 100;
```

The expression must yield a BOOLEAN result. The sequence of statements between the symbols REPEAT and UNTIL are executed one or more times. Every time the sequence is executed, the expression is evaluated. When the resulting value becomes true the REPEAT statement is completed.

WHILE STATEMENT

The WHILE statement specifies that a statement is to be executed as long as some condition is true.

WHILE Statement



The following is an example of the WHILE statement:

```
WHILE COUNT < 80 DO BEGIN
    READ(CHARACTER);
    COUNT := COUNT + 1
END; (* WHILE *)
```

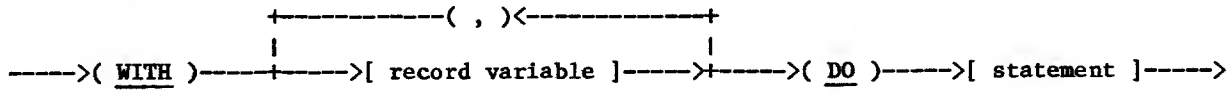
The expression must yield a result of type Boolean. The statement following DO will be executed zero or more times. The expression is evaluated before each execution.

The WHILE statement continues until the evaluation of the expression yields a false result. If the evaluation of the expression is false before execution of the WHILE statement, the statement following DO is not executed.

WITH STATEMENT

The WITH statement facilitates manipulation of record components.

WITH Statement



The following are examples of the WITH statement:

```
FOR I := 1 TO 100 DO
    WITH NAME[I] DO BEGIN
        IF (SEX = FEMALE) THEN FCOUNT := FCOUNT + 1;
        ELSE MCOUNT := MCOUNT + 1
    END; (* WITH *)

WITH HAND[1] DO BEGIN
    T := NORMAL;
    SUIT := CLUB;
    RANK := 8
END; (* WITH *)
```

WITH statements can be nested as in the following example:

```
WITH V1 DO S1
    WITH V2 DO S1
        .
        .
        .
        WITH Vn DO S1;
```

A shorter way to write the same nested WITH statement is the following:

```
WITH V1, V2, V3, ..., Vn DO S1;
```

The fields of the record variable(s) within the statement can be denoted by writing their field identifiers without preceding them with the denotation of the entire record variable.

The record variable selects a record; this selection cannot be changed in the statement. If the record variable has array indexes or pointers, changes to the array indexes or pointers within the WITH statement will not affect the selection.

A Pascal job usually passes through the following steps:

1. The source code (program) is compiled. The compiler generates either relocatable or absolute binary object code, and, if the L compiler option is selected, a listing of the source code.

2. The object code is loaded and linked with precompiled routines (for example, routines for input and output and routines predefined by the user).

3. The loaded code is executed.

You initiate these steps with the appropriate control statements. The following sequence shows the basic control statements to compile, load, and execute a program:

Batch Job Sequence

```
job.
USER, username, pwd.
CHARGE, chargenum, project.
PASCAL.  <---------- Step 1
LGO.  <------------- Steps 2 and 3
EOR
PROGRAM SAMPLE (INPUT, OUTPUT);
BEGIN
.
.
.
END.
EOR
  data
EOF
```

Interactive Job Sequence

```
file:
PROGRAM SAMPLE(INPUT,OUTPUT);
BEGIN
.
.
.
END.

PASCAL(I=file)  <--------  Step 1
LGO.    <-----------------  Steps 2 and 3
```

ORGANIZATION OF A COMPILED PROGRAM

The object code that is generated by the compiler is relocatable binary code separated into named logical records, or modules. Each module contains the code for a block in the program. The modules occur in the same order as the BEGIN/END blocks occur in the program. Global variables are placed in a separate module. The module names depend on the E compiler option. See the description of the E option under the heading Compiling a Program for an explanation of the entry-point names in the object code modules.

Here are two examples of source code and the object code they produce:

Source code

```
(*$E+*)
PROGRAM A(OUTPUT);
   PROCEDURE B;
      BEGIN (* B *)
      .
      .
      .
      END;
   PROCEDURE C;
      PROCEDURE D;
         BEGIN (* D *)
         .
         .
         .
         END;
         PROCEDURE E;
            BEGIN (* E *)
            .
            .
            .
            END;
      BEGIN (* C *)
      .
      .
      .
      END.
   BEGIN (* A *)
   .
   .
   .
   END.
```

Object code

Record:

1	B
2	D
3	E
4	C
5	A
6	A;

Source code

```
(*$E+*)
PROGRAM K(OUTPUT);
   PROCEDURE L;
      BEGIN (* L *)
      .
      .
      .
      END;
   PROCEDURE M;
   FORWARD;
   PROCEDURE N;
      BEGIN (* N *)
      .
      .
      .
      END;
   PROCEDURE M;
      BEGIN (* M *)
      .
      .
      .
      END;
   BEGIN (* K *)
   .
   .
   .
   END.
```

Object code

Record:

1	L
2	N
3	M
4	K
5	K;

COMPILING A PROGRAM

To initiate compilation of your program, use the control statement

```
PASCAL(I=sfn,L=lfn,B=bfn,GO,PD=pd,PS=ps,PL=pl/options)
```

where

sfn	Input source program file name. I alone defaults to source program name COMPILE. I omitted defaults to source program name INPUT.
lfn	Compiler listing file name. L=0 deselects the compiler listing (except errors). L omitted defaults to compiler listing file name OUTPUT.
bfn	Relocatable binary object file name. B omitted defaults to relocatable binary file name LGO.
GO	Selects automatic load and execute. GO omitted defaults to no execute.
pd	Print density of pd lines per inch; pd can be 6 or 8. PD omitted defaults to 6 lines per inch. If the PS parameter is used, then the PD parameter is ignored.
ps	Compiler listing page size of ps lines. If ps > 1000, paging is turned off; if ps < 20, page size is set to 20 lines. PS omitted reflects the value of the PD parameter.
pl	Program print limit of pl lines. The PL parameter applies only to the file variable OUTPUT of the program being compiled. PL or PL=0 selects the maximum line limit of MAXINT lines. PL omitted defaults to 2000 lines.
options	One or more compiler options.

The parameters sfn, lfn, and bfn are order-independent. The following control statement requests the compiler to compile source file SS and to produce both a program listing file named OUTPUT, the default program listing file, and a binary object file named BB.

```
PASCAL(I=SS,B=BB)
```

At least 50000 octal words of central memory are needed to run the compiler.

You can control the compilation mode with compiler directives. For example, you can request the compiler to insert or omit runtime test instructions with compiler directives.

Compiler directives are written as comments, but with a dollar sign ($) as the first character. For example, (*$T+*).

Compiler directives can be placed anywhere in your program, which enables you to activate options over specific parts of the program.

Each option consists of an option letter followed by the new value of the option setting. The value may be a plus sign (+) or a minus sign (-), which turns some options on and off like switches. Alternatively, the value may be a decimal or octal (indicated by a radix B) integer for numeric options, or a literal string for string options (see the E, I, and L options). The rules for these strings are the same as those for character strings that appear in any Pascal program. Finally, if the value is an equals sign (=), the option is set to its previous value (except with the I option). However, only one previous value is remembered.

Option scanning terminates when any entry that is not an appropriate option letter or option value is entered. For example, if you set a switch option to a numeric value, option scanning will end and no error messages will be produced (except with the I option). Errors also terminate option scanning.

The following options are available:

B Determines the size of a file buffer. If the value of B <= 64, the value of the B option acts as a buffer factor, and the actual buffer size (in words) is at least 128 times the buffer factor. If the value of B > 64, the value of the B option acts as the actual buffer size.

The compiler adds one to the value of B and then rounds the value to the next multiple of the file element size. Buffer sizes must be adjusted to fit the requirements of peripheral hardware devices. Disk files need at least B1 (or B128). Tape files need at least B4 (or B512).

The buffer size for a file is bound to its type. The type TEXT is predefined at the time the compiler reads the reserved symbol PROGRAM. Therefore, to change the buffer size for textfiles (including INPUT and OUTPUT), the value of the B option must be set prior to the program heading.

Default is B2.

E Allows you to control the entry-point names that are generated by the compiler for the main program, main variables block, procedures, functions, and labels. Entry points are required by the operating system loader. The E option is of special interest to you if you want to create a library of compiled, relocatable procedures and functions. The following paragraphs describe the effect of the E option:

a) Procedures and functions declared as EXTERN or FORTRAN get an entry-point name equal to the first seven characters of the procedure or function name. Other routines get an entry-point name that depends on the value of the E option when the routine name is analyzed:

E- Creates a unique entry-point name of the form PRCnnnn (where nnnn is an octal number from 0001 to 7777).

E+ Uses the first seven characters of the routine name as the entry-point name.

An extended form of the E option can be used to create an entry-point that is unrelated to the name of the routine. The following example illustrates the extended form for procedures and functions:

```
FUNCTION (*$E'P.RND'*) ROUND(X: REAL): REAL;
```

The entry-point for function ROUND is P.RND. The extended form of the E option allows you to define any entry-point that is accepted by the loader, even ones that include special characters, such as a period. The extended form of the E option applies equally to EXTERN, FORTRAN, and local routines. However, the E option must be specified between the word FUNCTION or PROCEDURE and the routine name.

b) The main program and main variables block get an entry-point name that depends on the value of the E option in the following way:

E- Uses P.MAIN as the main program and main variables block entry-point name.

E+ Uses the first seven characters of the program name as the main program entry-point name and the first six characters of the program name followed by a semicolon as the main variables block entry-point name.

You can use the extended form of the E option for the main program, but separate names should be specified for the main program block and the main variables block. For example,

```
PROGRAM (*$E'P.MAIN'/'P.VARS' *) MYPROG(OUTPUT);
```

c) Labels that are used in GOTO statements that exit a block are automatically assigned an entry-point name of the form PASCL.x (where x is a letter or digit). The entry-point name of any label can be explicitly assigned with the extended E option. In this case, the E option must immediately precede the declaration of the label. For example,

```
LABEL (*$E'L.1' *) 1,2, (*$E'L.LOOPS' *) 13;
```

It is your responsibility to ensure that duplicate entry-point names are not created when you specify the E- option. You must avoid creating duplicate entry-point names and must ensure that created entry-point names are acceptable to the system loader when you specify the extended form of the E option. The extended form of the E option exists mainly for the Pascal library.

This option can change the meaning of your program. The use of the E option results in a compiler warning message.

Default is E-.

I Controls the inclusion of external text. The I option includes source code from an external file. This directive has the following two forms:

```
(*$I'PACKAGE'/'FILE'*)

(*$I'PACKAGE'*)
```

The first form attempts to find an entry named PACKAGE on the file named FILE. The second form attempts to find the entry named PACKAGE on the default file, which is PASCLIB. The included text is not restricted to declarations; it can also contain full procedures and functions. Because the text entry is simply inserted into the text of your program, the include facility can be used to create full source libraries.

The included text is written on the program listing if L+ was selected, thereby giving you an accurate record of what was compiled. A complete record is important if you plan to transport the program to another implementation of Pascal. Compiler options embedded within included text will change previous option settings unless they are explicitly restored with an equal sign (=) in the text itself.

This option can change the meaning of your program. The use of the I option results in a compiler warning message.

L Controls the listing of the program text. The L option turns the listing on and off during compilation and sets page titles and subtitles. L+ turns the listing on and L- turns it off. L- is equivalent to L=0 on the PASCAL control statement. If L is followed by a character string, the page title or subtitle is set. The first such specification sets the main title, while subsequent specifications set the subtitle and cause a page eject. To set the title on the first printed page, the L option must appear on the first line.

Default is L+.

O Controls the effect of the compiler options specified in the source text of a program. O+ enables all compiler options within the source text and O- disables all compiler options within the source text. This option allows you to import a program and check it to see if it conforms to the standard by compiling it with S+,O-.

This option can change the meaning of your program. The use of the O option results in a compiler warning message.

Default is O+.

P Directs the compiler to generate a Post Mortem Dump (PMD) listing in the event of a runtime error. Pascal PMD is not related to FORTRAN PMD. P+ requests PMD to provide a description of each procedure or function that was active at the time of the error, including the line number of the statement that was currently being executed and the names and values of all of the unstructured local variables. Values of pointer variables are printed as 6-digit octal addresses, and values of ALFA variables are printed as 10-character strings. A value of UNDEF means undefined. P+ adds no execution time penalty and only a minimal storage penalty. P+ is recommended until you are sure that your program is correct. P- suppresses most of the PMD information; it includes enough information to list the name of the procedure in which the error occurred. P0 is an option setting designed especially for the Pascal compiler and library. Procedures compiled with P0 are transparent to PMD. Compiling an entire program with P0 deletes the minimal information (3 words per procedure), which includes the name of the procedure and the locations of the entry point and constants. P0 can be used for production programs to delete all unnecessary traceback information.

Default is P+.

R Controls reduce mode. R is used in conjunction with the W option to control execution field length.

Default is R+.

S Directs the compiler to issue nonfatal warning messages when a nonstandard Pascal extension is detected. S+ enables the listing of nonfatal warning messages, and S- disables the listing of nonfatal warning messages. The S option can be switched on and off anywhere within a Pascal program.

Default is S+.

T Directs the compiler to generate extra code that can be used to perform runtime tests to check the following:

a) That the index used for array-indexing operations lies within the specified array bounds.

b) That the value that is assigned to a variable of a subrange type lies within the specified range. This check is also performed when reading such variables.

c) That no divide-by-zero operations were performed.

d) That the absolute value of the result of an automatic real-to-integer conversion is less than MAXINT (2**48 - 1).

e) That there was no overflow or underflow from a real expression.

f) That the evaluated expression in a CASE statement corresponds to a constant in a case list element (unless OTHERWISE is used).

g) That p is a valid pointer when it is referenced as p↑ or DISPOSE(p). The T+ option must be selected when the pointer type is declared and when the pointer is referenced.

h) That SET elements are within the declared range after assignments to set variables are made.

Also, the control variable in all FOR statements is set to an undefined value upon normal exit from the statement if T+ is selected. T+ adds a severe execution time penalty. T+ is recommended until you are sure that your program is correct.

Default is T+.

U Restricts the number of characters that are scanned by the compiler in every source line. U+ restricts the number of characters to 72. The restriction is convenient when you use the default widths with the UPDATE or MODIFY text maintenance programs. U- sets the number of relevant characters to 120. U may be set to any specific numeric value between 10 and 120. The remainder of the line (past the width specified by this option) is treated as a comment. The U option is best used on the first line of the Pascal source program.

This option can change the meaning of your program. The use of the U option results in a compiler warning message.

Default is U-.

W Controls the workspace size. W can be used in conjunction with the R option to control runtime field length.

Wn sets the number of words to be used for the work space (where n is a string of digits with an optional post-radix B).

W0 requests the Pascal compiler to calculate an appropriate work space size. The compiler sums the lengths of all nonglobal variables declared in the program, then adds a safety factor of 2000 octal (1024 decimal) words. The value that the compiler estimates for the W option is printed at the bottom of the compiler listing.

Default is W0.

X Determines the number of X registers used for passing parameter descriptors. If the value of the X option is in the range ($0 \leq N \leq 5$), the first N parameter descriptors are passed in the registers X0 to X(N-1) (the first in X0, the second in X1, and so on). Extra parameters are passed through a table in memory.

N > 0 reduces the size of the code produced by the compiler and usually decreases the execution time. However, you must be aware that with the Ith parameter and with N > 0, the compiler cannot use registers X0 to XJ (where J is the minimum of (N-1) and (I-2)) for its computation. It is possible for the compiler to give the message: EXPRESSION TOO COMPLICATED where N > 0.

Default is X4.

OVERVIEW OF THE RUNTIME SYSTEM

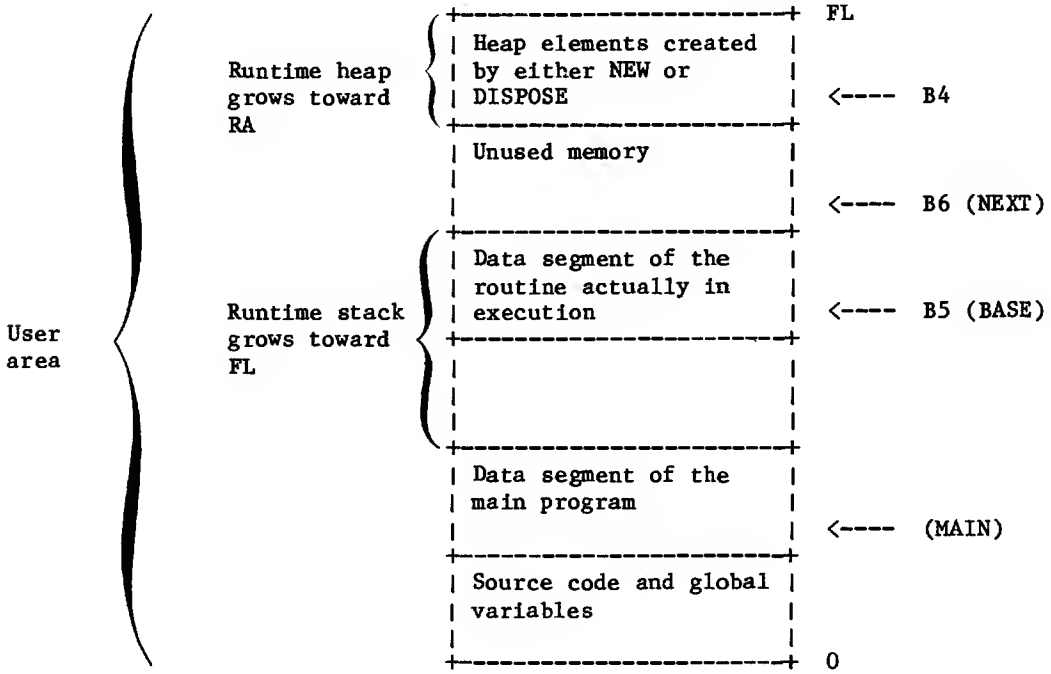
Code and data are separated from each other at runtime. The local data from each executed routine is united in a data segment and is addressed by an offset that is relative to the segment origin, which is called the base address. At runtime, a stack containing the data segments of all executed routines is provided. Because the base addresses of the data segments vary during runtime, variable addressing is nontrivial. However, this way of organizing data guarantees maximum storage economy. Every data segment exists only during the routine execution; the data segment is created at routine entry and discarded at routine exit.

Data segment stacking and unstacking requires a dynamic link (DL). The dynamic link chains each data segment to its immediate predecessor in the stack. Variable addressing requires a static link (SL). The static link chains those data segments that are currently accessible. DL and SL are incorporated in the head of every data segment.

For example, refer to the following source code:

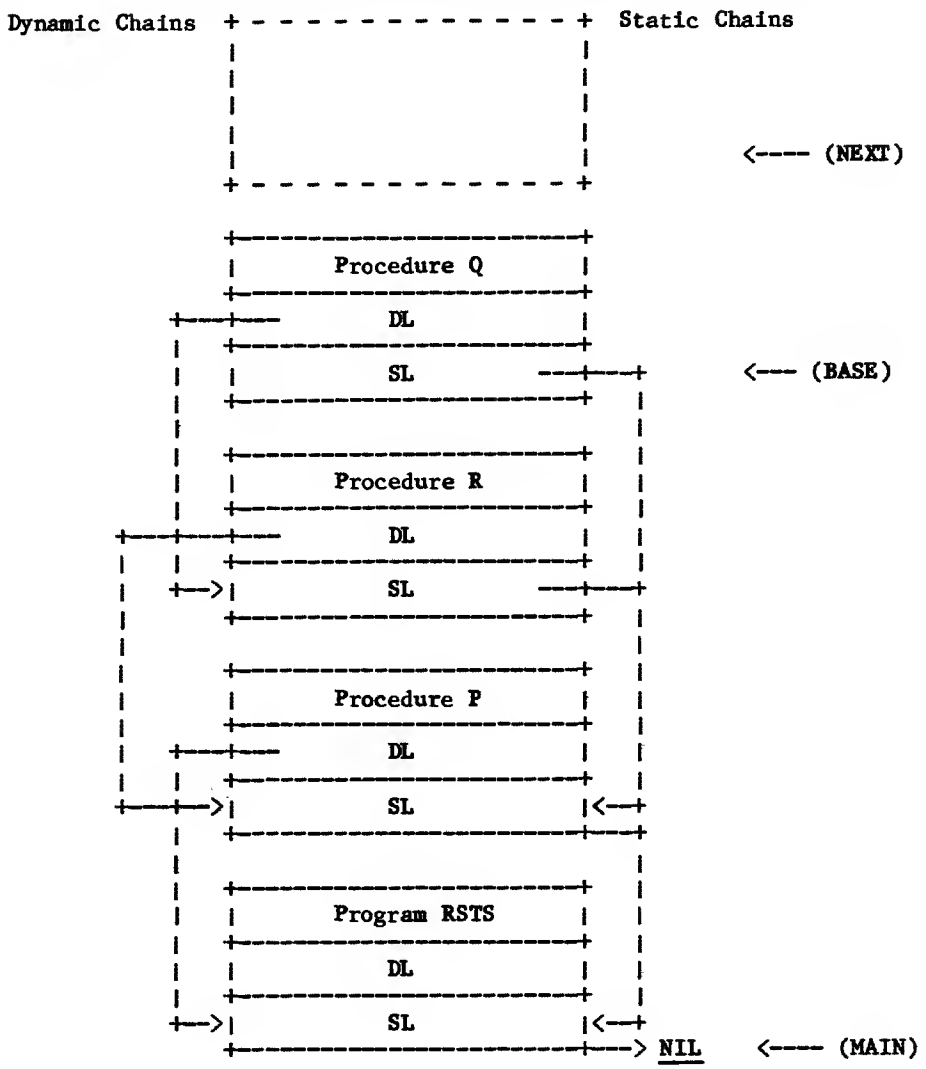
```
(*$E+*)
PROGRAM RSTS(OUTPUT);
PROCEDURE P;
    PROCEDURE Q;
    BEGIN (* Q *)
    .
    .
    .
    END;
    PROCEDURE R;
    BEGIN (* R *)
    .
    .
    .
    Q;
    .
    .
    .
    END;
BEGIN (* P *)
.
.
.
R;
.
.
.
END;
BEGIN (* RSTS *)
.
.
.
P;
.
.
.
END.
```

The following diagram shows the stack of data segments that corresponds to program RSTS. The stack, which grows upward, originates from the calling sequence: RTST -> P -> R -> Q. BASE is the base address of the most recently created data segment. BASE is the head of the chains. NEXT defines the base address of the next data segment to be stacked.



If the runtime heap and runtime stack ever meet during execution, then the error message RUNTIME STACK OVERFLOW is issued and the program is aborted. You must increase the runtime field length with the RFL command (for example, RFL,70000) and then rerun your program.

The following diagram shows the static and dynamic links between the data segments:



LOADING AND EXECUTING A PROGRAM

To initiate loading and execution of your program, use the control statement:

```
OC(f1,f2, ... ,fn)
```

where

OC The file that contains the object code, or relocatable binary code.

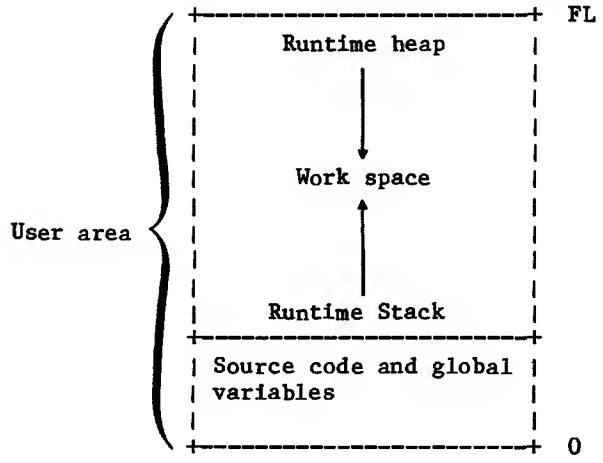
fi The names of files that contain routines that are external to the program, but that are used during execution.

Routines that are referenced, but not included in file OC, are searched for in the PASCLIB system library.

Routines that are referenced, but not included either in the file OC or in the PASCLIB system library, are searched for in the system library and in any global user libraries. See the Loader Version 1 Reference Manual for more information about libraries and library searching.

You can load and initiate execution in several other ways.

After the loading process is complete, a contiguous piece of unused memory remains at the upper end of the user area. This area, called the work space, is used for the runtime stack and runtime heap during execution. The runtime stack grows upward from the lower end while the runtime heap grows downward from the upper end.



The W compiler option controls the calculation of the work space (WS) value.

Wn sets the number of words to be used for the WS (n is a string of digits with an optional post-radix B).

W0 requests the Pascal compiler to calculate an appropriate work space size. Pascal sums the lengths of all nongobal variables declared in the program, then adds a safety factor of 2000 octal (1024 decimal) words. The value that the compiler estimates for the W option is printed at the bottom of the compiler listing.

The R compiler option controls what is done with the work space value. R+ requests that the user program be given the right amount of memory for both the code including global variables (CS) and the WS, even if this is a reduction. R- requests that the memory be increased only if it is necessary to satisfy the sum of the CS and WS. In other words, the memory allocation will never be decreased if R- is set. This option has an effect that is analogous to the REDUCE control statement.

The default setting is W0,R+. This setting causes the compiler to calculate the WS value and request memory allocation equal to the WS value, regardless of whether or not an increase or decrease is required. This setting will always allocate enough memory for programs that do not use recursion or dynamic allocation, which is the case for most programs. For some programs, however, the default setting may not be appropriate.

When you set the work space value explicitly, you should note that there is hidden data (temporary space for anonymous variables) that is used by the Pascal program itself. Therefore, you should increase your WS estimate to provide a margin of safety. A good rule of thumb is to add about 10 words per procedure plus an additional several hundred words.

UNDERSTANDING RUNTIME ERROR MESSAGES

When a runtime error occurs, a dayfile message explaining the error is given together with a Post Mortem Dump.

SAMPLE PROGRAMS 8

This section poses some problems and provides one or more solutions to the problem.

The first problem deals with placing a class of three steers.

In a judging contest, the official judges the steers on qualities such as height, straightness along the back, and amount of muscle. The steers are numbered 1, 2, and 3, so it is possible for the official to determine the correct placing as: 3, 1, 2.

After the official determines the correct placing, students judge the same class to determine what they feel is the correct placing (the official's placing is unknown to the students).

A student can place the class as one of the following combinations:

```
3   1   2
3   2   1
1   3   2
2   1   3
2   3   1
1   2   3
```

A perfect match between the official's and a student's placing is awarded 50 points. A student whose placing does not match the official's is penalized for each incorrect decision that was made. The penalty is calculated using a number called the degree of difficulty or cut. The cut between a pair of steers is also determined by the official. An example of a cut assignment is:

```
Official placing:  3   1   2

Cuts:                5   1
```

If the official assigns a cut of 5 between steers 3 and 1, then there is a clear difference in quality in the two steers; switching the placing of this pair results in a large penalty. If the official assigns a cut of 1 between steers 1 and 2, then there is a small difference in quality in the two steers; switching the placing of this pair results in a lesser penalty. The following are sample penalty calculations:

```
Official placing:  3   1   2

Cuts:                5   1

Student placing:   1   3   2
```

The score would be calculated as 50 - 5 = 45 because the top pair was switched.

```
Official placing:  1   2   3

Cuts:                1   3

Student placing:   3   2   1
```

The score would be calculated as 50 - (2*cut1 + 2*cut2) = 42 because the top and bottom placing was switched.

The problem is to write a Pascal program that accepts as input the official's placing, cuts, and student's placing, calculates the score, and outputs the score.

The first solution uses arrays to hold the data, IF statements to perform the calculations, and labeled statements to control the flow of execution.

```
PROGRAM JUDGE(INPUT/,OUTPUT);
TYPE
    PLACINGS = ARRAY[1..3] OF INTEGER;
    CUTS = ARRAY[1..2] OF INTEGER;
VAR
    O,J : PLACINGS;
    CUT : CUTS;
    I,RESULT : INTEGER;
LABEL
    50,75;
BEGIN
(* INPUT OFFICIAL PLACING. *)
    WRITELN('INPUT OFFICIAL PLACING');
    FOR I := 1 TO 3 DO READ(O[I]);
(* INPUT CUTS. *)
    WRITELN('INPUT OFFICIAL CUTS');
    FOR I := 1 TO 2 DO READ(CUT[I]);
(* INPUT JUDGE'S PLACING OR ZERO. *)
50 : WRITELN('INPUT JUDGES'' PLACING OR FOUR ZEROS');
    FOR I := 1 TO 3 DO READ(J[I]);
    IF (J[1] = 0) THEN GOTO 75;
(* BEGIN CALCULATION OF SCORE.  PERFECT SCORE. *)
    IF ((O[1]=J[1]) AND (O[2]=J[2]))
        THEN RESULT := 50;
(* TOP AND BOTTOM PAIR SWITCHES. *)
    IF ((O[1]=J[2]) AND (O[2]=J[1]))
        THEN RESULT := 50 - CUT[1];
    IF ((O[2]=J[3]) AND (O[3]=J[2]))
        THEN RESULT := 50 - CUT[2];
(* TOP TO BOTTOM. *)
    IF ((O[1]=J[3]) AND (O[2]=J[1]))
        THEN RESULT := 50 - (2*CUT[1] + CUT[2]);
(* SIMPLE BUST. *)
    IF ((O[1]=J[2]) AND (O[2]=J[3]))
        THEN RESULT := 50 - (CUT[1] + 2*CUT[2]);
(* MAJOR BUST. *)
    IF ((O[1]=J[3]) AND (O[2]=J[2]))
        THEN RESULT := 50 - (2*CUT[1] + 2*CUT[2]);
(* OUTPUT SCORE. *)
    WRITELN('SCORE IS ',RESULT:2);
    GOTO 50;
75 : WRITELN('END OF PROGRAM')
END.
```

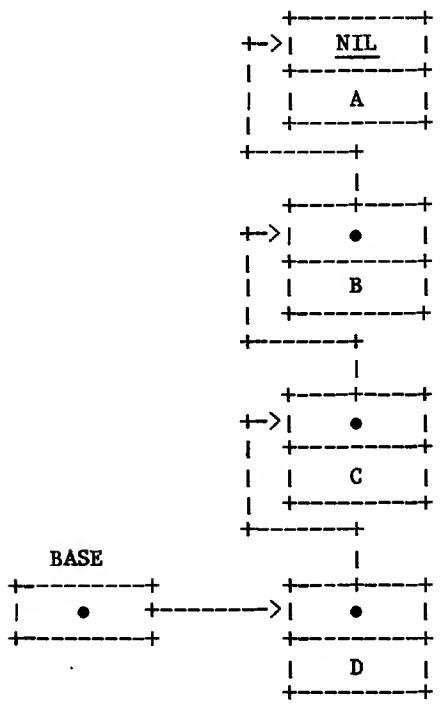
The second solution decodes the student's placing to match the placing 1 2 3 using WHILE and REPEAT statements and then calculates the penalty using a CASE statement and a function. The use of WHILE or REPEAT statements to control execution of a program is preferred over the use of labeled statements because the result is a more structured program.

```
PROGRAM JUDGE(INPUT/,OUTPUT);
TYPE
    PLACINGS = ARRAY[1..3] OF INTEGER;
    CUTS = ARRAY[1..2] OF INTEGER;
VAR
    O,J,R : PLACINGS;
    CUT : CUTS;
    I,M,N,SCORE : INTEGER;
FUNCTION RESULT(X,Y : INTEGER) : INTEGER;
    BEGIN
        RESULT := 50 - (X*CUT[1] + Y*CUT[2])
    END;
BEGIN
(* INPUT OFFICIAL PLACING. *)
    WRITELN('INPUT OFFICIAL PLACING');
    FOR I := 1 TO 3 DO READ(O[I]);
(* INPUT CUTS. *)
    WRITELN('INPUT OFFICIAL CUTS');
    FOR I := 1 TO 2 DO READ(CUT[I]);
(* INPUT JUDGE'S PLACING. *)
    WRITELN('INPUT JUDGES'' PLACING');
    FOR I := 1 TO 3 DO READ(J[I]);
(* CREATE ARRAY R AS IF OFFICIAL PLACING WERE 1 2 3. *)
    FOR N := 1 TO 3 DO BEGIN
        M := 0;
        REPEAT
            M := M + 1;
        UNTIL J[M] = O[N];
        R[M] := N
    END; (* FOR *)
(* CALCULATE RESULT. *)
    CASE (100*R[1] + 10*R[2] + R[3]) OF
        123 : SCORE := RESULT(0,0);
        132 : SCORE := RESULT(0,1);
        213 : SCORE := RESULT(1,0);
        231 : SCORE := RESULT(2,1);
        312 : SCORE := RESULT(1,2);
        321 : SCORE := RESULT(2,2)
    END; (* CASE *)
(* OUTPUT SCORE. *);
    WRITELN('SCORE IS ',SCORE:2);
    WRITELN('END OF PROGRAM')
END.
```

The second problem deals with building a linked list. The following program creates a last-in-first-out (LIFO) linked list of four nodes. The data area in each node is assigned a character in the alphabet. After the linked list is constructed, it is traversed from the last entry to the first entry. Traversal is verified by writing the contents of the data area in each node.

```
PROGRAM LNKLIST(INPUT/,OUTPUT);
TYPE
    POINTER = ↑NODE;
    NODE =  RECORD
                NEXTPNTR : POINTER;
                DATA : CHAR
            END;
VAR
    BASE,PNTR : POINTER;
    I : INTEGER;
BEGIN
(* CREATE A POINTER THAT POINTS TO NIL. *)
    BASE := NIL;
(* CREATE NODES AND LINK THEM. *)
    FOR I := 1 TO 4 DO BEGIN
    (* CREATE A NEW NODE. *)
        NEW(PNTR);
    (* PUT DATA INTO THE NODE DATA AREA. *)
        READLN(PNTR .DATA);
    (* PUT THE BASE POINTER VALUE INTO THE NODE POINTER. *)
        PNTR↑.NEXTPNTR := BASE;
    (* POINT THE BASE POINTER TO THE NODE. *)
        BASE := PNTR
    END; (* FOR *)
    PNTR↑:= BASE;
    WHILE PNTR <> NIL DO BEGIN
    (* VERIFY ORDER OF NODES. *)
        WRITELN(PNTR↑.DATA);
    (* POINT TO THE NEXT NODE. *)
        PNTR := PNTR↑.NEXTPNTR
    END; (* WHILE *)
    WRITELN('END OF PROGRAM')
END.
```

If you insert A, B, C, D as data for the nodes, the resulting linked list would appear as follows:

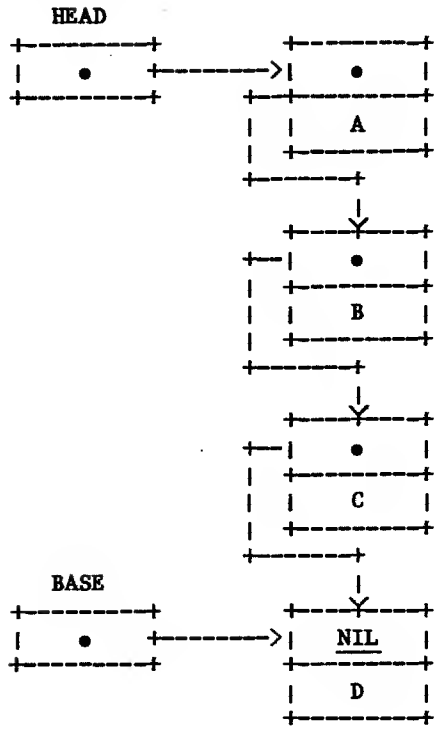


The following program is a variation of the linked list program. A linked list of four nodes is again created, but the first node is pointed to by a pointer named HEAD and the last node by a pointer named TAIL. The advantage of creating the list this way is that modifying the list is much easier.

The list must contain at least one node.

```
PROGRAM HEADTAIL(INPUT/,OUTPUT);
TYPE
    POINTER = ↑NODE;
    NODE =  RECORD
                NEXTPNTR : POINTER;
                DATA : CHAR
            END;
VAR
    HEAD,TAIL,PNTR : POINTER;
    I : INTEGER;
BEGIN
(* CREATE FIRST NODE AND POINT HEAD AND TAIL TO IT. *)
    NEW(PNTR);
    READLN;
    READ(PNTR .DATA);
    PNTR↑.NEXTPNTR := NIL;
    HEAD := PNTR;
    TAIL := PNTR;
(* CREATE OTHER THREE NODES. *)
    FOR I := 1 TO 3 DO BEGIN
        NEW(PNTR);
        READLN;
        READ(PNTR↑.DATA);
        PNTR↑.NEXTPNTR := TAIL↑.NEXTPNTR;
        TAIL↑.NEXTPNTR := PNTR↑.NEXTPNTR;
        TAIL := PNTR
    END; (* FOR *)
(* VERIFY ORDER OF NODES. *)
    PNTR := HEAD;
    REPEAT
        WRITELN(PNTR↑.DATA);
        PNTR := PNTR↑.NEXTPNTR
    UNTIL PNTR↑.NEXTPNTR = NIL;
    WRITELN(PNTR↑.DATA);
    WRITELN('END OF PROGRAM')
END.
```

If you insert A, B, C, D as data for the nodes, the resulting linked list would appear as follows:



CHARACTER SETS A

Table A-1 shows the character correspondence between the internal Pascal character set and the CDC Scientific and CDC ASCII character sets.

All program statements in this manual are shown in the internal Pascal character representation. You must translate this representation into the character set used at your site.

Table A-1. Characters Sets

Ordinal Number	Pascal Character	CDC Scientific Character Set	CDC ASCII Character Set
0	Undefined	End of Line in 63 : (colon) in 64	End of Line in 63 : (colon) in 64
1	A	A	A
2	B	B	B
3	C	C	C
4	D	D	D
5	E	E	E
6	F	F	F
7	G	G	G
8	H	H	H
9	I	I	I
10	J	J	J
11	K	K	K
12	L	L	L
13	M	M	M
14	N	N	N
15	O	O	O
16	P	P	P
17	Q	Q	Q
18	R	R	R
19	S	S	S

(Continued on next page)

Table A-1. Characters Sets

(Continued)

Ordinal Number	Pascal Character	CDC Scientific Character Set	CDC ASCII Character Set
20	T	T	T
21	U	U	U
22	V	V	V
23	W	W	W
24	X	X	X
25	Y	Y	Y
26	Z	Z	Z
27	0	0	0
28	1	1	1
29	2	2	2
30	3	3	3
31	4	4	4
32	5	5	5
33	6	6	6
34	7	7	7
35	8	8	8
36	9	9	9
37	+	+	+
38	-	-	-
39	*	*	*
40	/	/	/
41	(	(	(
42	)	)	)
43	$	$	$
44	=	=	=
45	(space)	(space)	(space)
46	, (comma)	, (comma)	, (comma)

(Continued on next page)

Table A-1. Characters Sets

Ordinal Number	Pascal Character	CDC Scientific Character Set	CDC ASCII Character Set
47	. (period)	. (period)	. (period)
48	# (number sign)	≡ (equivalence)	# (number sign)
49	[(left bracket)	[(left bracket)	[(left bracket)
50	] (right bracket)	] (right bracket)	] (right bracket)
51	: (colon)	: (colon) in 63 % (percent) in 64	: (colon) in 63 % (percent) in 64
52	" (quote)	<> (not equal)	" (quote)
53	_ (underline)	(* (open comment)	_ (underline)
54	! (exclamation)	OR (logical OR)	! (exclamation)
55	& (ampersand)	AND (logical AND)	& (ampersand)
56	' (apostrophe)	↑ (up arrow)	' (apostrophe)
57	? (question)	*) (close comment)	? (question)
58	< (less than)	< (less than)	< (less than)
59	> (greater than)	> (greater than)	> (greater than)
60	@ (commercial at)	<= (less equal)	@ (commercial at)
61	\ (back slash)	>= (greater equal)	\ (back slash)
62	↑ (up arrow)	NOT (logical NOT)	^ (circumflex)
63	; (semicolon)	; (semicolon)	; (semicolon)

Some symbols have alternative representations that can be used interchangeably with the symbol. The symbols and their alternative representations are:

Symbol	Aternative Symbol
|	@
[	(.
]	.)

COMPILATION ERROR MESSAGES B

The compiler indicates an error by printing an arrow that points to the place in the text where the error is detected. This is not always the place where the error is made. The arrow is followed by a number, which indicates what kind of error was detected. A list of numbers used in error messages and their corresponding messages is given at the end of the compilation. The list is also given on the file containing the compiler listing.

At most 10 errors will be indicated on one line.

ERRORS THAT ARE DETECTED

```
  1:   ERROR IN SIMPLE TYPE.
  2:   IDENTIFIER EXPECTED.
  3:   "PROGRAM" EXPECTED.
  4:   ")" EXPECTED.
  5:   ":" EXPECTED.
  6:   UNEXPECTED SYMBOL.
  7:   ERROR IN PARAMETER LIST.
  8:   "OF" EXPECTED.
  9:   "(" EXPECTED.
 10:   ERROR IN TYPE.
 11:   "[" EXPECTED.
 12:   "]" EXPECTED.
 13:   "END" EXPECTED.
 14:   ";" EXPECTED.
 15:   INTEGER CONSTANT EXPECTED.
 16:   "=" EXPECTED.
 17:   "BEGIN" EXPECTED.
 18:   ERROR IN DECLARATION PART.
 19:   ERROR IN FIELD-LIST.
 20:   "," EXPECTED.
 21:   ".." EXPECTED.
 40:   VALUE PART ALLOWED ONLY IN MAIN PROGRAM.
 41:   TOO FEW VALUES SPECIFIED.
 42:   TOO MANY VALUES SPECIFIED.
 43:   VARIABLE INITIALIZED TWICE.
 44:   TYPE IS NEITHER ARRAY NOR RECORD.
 45:   REPETITION FACTOR MUST BE GREATER THAN ZERO.
 50:   ERROR IN CONSTANT.
 51:   ":=" EXPECTED.
 52:   "THEN" EXPECTED.
 53:   "UNTIL" EXPECTED.
 54:   "DO" EXPECTED.
 55:   "TO" OR "DOWNTO" EXPECTED.
 57:   "FILE" EXPECTED.
 58:   ERROR IN FACTOR.
 59:   ERROR IN VARIABLE.
 60:   FILE TYPE IDENTIFIER EXPECTED.
101:   IDENTIFIER DECLARED TWICE.
102:   LOWBOUND EXCEEDS HIGHBOUND.
103:   IDENTIFIER IS NOT OF APPROPRIATE CLASS.
104:   IDENTIFIER NOT DECLARED.
105:   SIGN NOT ALLOWED.
106:   NUMBER EXPECTED.
107:   INCOMPATIBLE SUBRANGE TYPES.
108:   FILE NOT ALLOWED HERE.
```

109: TYPE MUST NOT BE REAL.
110: TAGFIELD TYPE MUST BE SCALAR OR SUBRANGE.
111: CONSTANT IS INCOMPATIBLE WITH TAG TYPE OR OUT OF RANGE.
112: INDEX TYPE MUST NOT BE REAL.
113: INDEX TYPE MUST BE SCALAR OR SUBRANGE.
114: BASE TYPE MUST NOT BE REAL.
115: BASE TYPE MUST BE SCALAR OR SUBRANGE.
116: ERROR IN TYPE OF PREDECLARED PROCEDURE PARAMETER.
117: UNSATISFIED FORWARD REFERENCE.
119: FORWARD DECLARED; REPETITION OF PARAMETER LIST NOT ALLOWED.
120: FUNCTION RESULT TYPE MUST BE SCALAR, SUBRANGE OR POINTER.
121: FILE VALUE PARAMETER NOT ALLOWED.
122: FORWARD-DECLARED FUNCTION; REPETITION OF RESULT TYPE NOT ALLOWED.
123: MISSING RESULT TYPE IN FUNCTION DECLARATION.
124: FIXED-POINT FORMATTING ALLOWED FOR REALS ONLY.
125: ERROR IN TYPE OF PREDECLARED FUNCTION PARAMETER.
126: NUMBER OF PARAMETERS DOES NOT AGREE WITH DECLARATION.
127: ALL PARAMETERS IN A GROUP MUST HAVE THE SAME TYPE.
128: PARAMETER PROCEDURE/FUNCTION IS NOT COMPATIBLE WITH DECLARATION.
129: TYPE CONFLICT OF OPERANDS.
130: EXPRESSION IS NOT OF SET TYPE.
131: ONLY EQUALITY TESTS ALLOWED.
132: "<" AND ">" NOT ALLOWED FOR SET OPERANDS.
133: FILE COMPARISON NOT ALLOWED.
134: INCORRECT TYPE OF OPERAND(S).
135: TYPE OF OPERAND MUST BE BOOLEAN.
136: SET ELEMENT MUST BE SCALAR OR SUBRANGE.
137: SET ELEMENT TYPES NOT COMPATIBLE.
138: TYPE OF VARIABLE IS NOT ARRAY.
139: INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION.
140: TYPE OF VARIABLE IS NOT RECORD.
141: TYPE OF VARIABLE MUST BE FILE OR POINTER.
142: INCORRECT PARAMETER SUBSTITUTION.
143: INCORRECT TYPE OF FOR-STATEMENT CONTROL VARIABLE.
144: INCORRECT TYPE OF EXPRESSION.
145: TYPE CONFLICT.
146: ASSIGNMENT OF FILES NOT ALLOWED.
147: INCORRECT TYPE OF CASE CONSTANT.
148: SUBRANGE BOUNDS MUST BE SCALAR.
149: INDEX TYPE MUST NOT BE INTEGER.
150: ASSIGNMENT TO THIS FUNCTION IS NOT ALLOWED.
151: ASSIGNMENT TO FORMAL FUNCTION IS NOT ALLOWED.
152: NO SUCH FIELD IN THIS RECORD.
155: CONTROL VARIABLE MUST BE LOCAL.
156: CASE CONSTANT APPEARS TWICE.
157: RANGE OF CASE CONSTANTS IS TOO LARGE (MAX = 2001).
158: MISSING CORRESPONDING VARIANT DECLARATION.
159: REAL OR STRING TAGFIELDS NOT ALLOWED.
160: PREVIOUS DECLARATION WAS NOT FORWARD.
161: MULTIPLE FORWARD DECLARATION.
162: MISSPELLED RESERVED WORD OR UNRECOGNIZED DIRECTIVE.
163: VALUE OF LABEL IS TOO LARGE (>9999).
164: PREDECLARED PROCEDURE/FUNCTION SUBSTITUTION NOT ALLOWED.
165: MULTIDEFINED LABEL.
166: MULTIDECLARED LABEL.
167: UNDECLARED LABEL.
168: UNDEFINED LABEL IN THE PREVIOUS BLOCK.
169: ERROR IN BASE SET.
170: VALUE PARAMETER EXPECTED.
172: UNDECLARED EXTERNAL FILE.
173: FORTRAN PROCEDURE OR FUNCTION EXPECTED.
174: PASCAL PROCEDURE OR FUNCTION EXPECTED.
175: MISSING FILE "INPUT" IN PROGRAM HEADING.

176: MISSING FILE "OUTPUT" IN PROGRAM HEADING.
177: ASSIGNMENT TO FUNCTION ALLOWED ONLY IN FUNCTION BLOCK.
178: MULTIDEFINED RECORD VARIANT.
179: X-OPTION OF ACTUAL PROCEDURE/FUNCTION DOES NOT MATCH FORMAL DECLARATION.
180: CONTROL VARIABLE MUST NOT BE FORMAL.
183: SUBRANGE OF TYPE REAL IS NOT ALLOWED.
184: INCORRECT USE OF FOR-STATEMENT CONTROL VARIABLE.
185: FUNCTION MUST BE ASSIGNED SOMEWHERE IN ITS BLOCK.
186: ONE OR MORE TAG VALUES ARE NOT SPECIFIED.
187: TAG FIELD IS NOT ALLOWED AS ACTUAL VARIABLE PARAMETER.
188: LABEL IS NOT ACCESSIBLE FROM HERE.
189: LABEL IS NOT ACCESSIBLE TO PREVIOUS GOTO(S) THAT USED IT.
190: IDENTIFIER BEING DEFINED WAS USED ALREADY IN THIS SCOPE.
191: TYPE IDENTIFIER EXPECTED.
192: CONTROL VARIABLE IS THREATENED BY A NESTED PROCEDURE OR FUNCTION.
193: ALREADY A CONTROL VARIABLE FOR AN ENCOMPASSING FOR-STATEMENT.
198: ALTERNATE INPUT NOT FOUND.
199: ONLY ONE ALTERNATE INPUT MAY BE ACTIVE.
201: ERROR IN REAL CONSTANT: DIGIT EXPECTED.
202: STRING CONSTANT MUST BE CONTAINED ON A SINGLE LINE.
203: INTEGER CONSTANT EXCEEDS RANGE.
204: 8 OR 9 IN OCTAL NUMBER.
205: STRINGS OF LENGTH ZERO ARE NOT ALLOWED.
206: INTEGER PART OF REAL CONSTANT EXCEEDS RANGE.
207: REAL CONSTANT EXCEEDS RANGE.
220: ONLY THE LAST DIMENSION MAY BE PACKED.
221: TYPE-IDENTIFIER OR CONFORMANT-ARRAY SCHEMA EXPECTED.
222: BOUND-IDENTIFIER EXPECTED.
223: ORDINAL-TYPE IDENTIFIER EXPECTED.
224: PACK AND UNPACK ARE NOT IMPLEMENTED FOR CONFORMANT ARRAYS.
240: MULTI-WORD VALUE PARAMETERS FOR FORTRAN ROUTINES MUST BE PASSED BY VAR DUE TO IMPL. RESTRICTION.
250: IDENTIFIER SCOPES TOO DEEPLY NESTED (MAX = 20).
251: TOO MANY NESTED PROCEDURES AND/OR FUNCTIONS (MAX = 10).
252: TOO MANY IDENTIFIER SCOPES (MAX = 131071).
255: TOO MANY ERRORS ON THIS SOURCE LINE (MAX = 9).
256: TOO MANY EXTERNAL REFERENCES (MAX = 4095).
259: EXPRESSION TOO COMPLICATED.
260: TOO MANY EXIT LABELS (MAX = 36).
261: TOO MANY LARGE VARIABLES.
262: NODE TO BE ALLOCATED IS TOO LARGE.
263: TOO MANY PROCEDURE/FUNCTION PARAMETERS (MAX = 77).
264: TOO MANY PROCEDURES AND FUNCTIONS (MAX = 4095).
300: DIVISION BY ZERO.
301: MOD BY NEGATIVE MODULO.
302: INDEX EXPRESSION OUT OF BOUNDS.
303: VALUE TO BE ASSIGNED IS OUT OF BOUNDS.
304: ELEMENT EXPRESSION OUT OF RANGE.
305: FIELD WIDTH MUST BE GREATER THAN ZERO.
320: WARNING--THIS PREDECLARED IDENTIFIER IS NON-STANDARD.
321: WARNING--OCTAL REPRESENTATION IS NON-STANDARD.
322: WARNING--INTEGER > MAXINT IS NON-STANDARD.
323: WARNING--SEGMENTED FILE IS NON-STANDARD.
324: WARNING--VALUE DECLARATION PART IS NON-STANDARD.
325: WARNING--EXTERNAL OR FORTRAN PROCEDURE IS NON-STANDARD.
326: WARNING--SECOND PARAMETER IS NON-STANDARD.
327: WARNING--ORD OF REAL OR POINTER IS NON-STANDARD.
328: WARNING--OTHERWISE IS NON-STANDARD.
329: WARNING--"+" OR "/" IN PROGRAM HEADING IS NON-STANDARD.
330: WARNING--MIXED ORDER OF DECLARATIONS IS NON-STANDARD.
331: WARNING--THIS OPTION MAY ALTER THE MEANING OF THE PROGRAM.
332: WARNING--CONFORMANT-ARRAY AS ACTUAL VALUE PARAMETER IS NON-STANDARD.
333: WARNING--TREATING A CONFORMANT-ARRAY PARAMETER AS A STRING IS NON-STANDARD.

350: WARNING--DIAGNOSTIC LANGUAGE SELECTED IS NOT AVAILABLE: ENGLISH WILL BE USED.
351: ARRAY TYPE IDENTIFIER EXPECTED.
352: ARRAY VARIABLE EXPECTED.
353: POSITIVE INTEGER CONSTANT EXPECTED.
394: COMPARISON OF DYNAMIC PARAMETERS NOT ALLOWED.
395: ASSIGNMENT TO/FROM DYNAMIC PARAMETER NOT ALLOWED.
396: MULTI-WORD VALUE PARAMETERS ARE NOT IMPLEMENTED FOR FORTRAN ROUTINES.
397: PACK AND UNPACK ARE NOT IMPLEMENTED FOR DYNAMIC ARRAYS.
398: IMPLEMENTATION RESTRICTION.

ERRORS THAT ARE NOT DETECTED

- Use of an integer factor in an expression that results in overflow. For example, an integer factor in an expression that results in a value greater than MAXINT.
- Use of uninitialized variables in expressions.
- Use of DISPOSE(p) when p↑ is a dynamic variable and a reference to p↑ exists.
- Use of a file variable f when a reference to the associated file buffer variable f↑ exists.
- Use of an undefined field in a variant record variable p↑, which is caused by using the form of NEW that has explicit tagfield constants. For example, NEW(p,c1, ..., cn).
- Use of an undefined function result in an expression.
- Use of READ when an integer or the integer part of a real number is greater than MAXINT.
- Access of or reference to a nonexistent field in a record variant.

GLOSSARY C

ABS -
An identifier that is associated in the Pascal language with the function that takes the absolute value of the argument.

Absolute Value -
A number that has been stripped of its sign.

Actual Parameter -
A variable that appears within parentheses and follows either the function identifier in a function call or the procedure identifier in a procedure call. An actual parameter is passed to either a function or a procedure.

ALFA -
An identifier that is predefined in the Pascal language as PACKED ARRAY[1..10] OF CHAR.

ARCTAN -
An identifier that is associated in the Pascal language with the function that takes the arctangent of the argument.

Argument -
An expression that is bound by parentheses and is operated on by a function or procedure. An argument can consist of multiple expressions, all of which are bound by a single pair of parentheses. Also called a parameter.

Array -
A set of elements that is identified by a single name.

ARRAY Type -
A structured data type that describes the number and type of the elements in an array.

Base Type -
Describes the scalar data type in a SET definition.

Binding -
The association of an actual parameter to a formal parameter during compilation.

Block -
A group of statements that is bound by a BEGIN and an END statement.

Boolean Literal -
Either of the predefined identifiers TRUE or FALSE.

BOOLEAN Type -
A simple data type that consists of the set [TRUE, FALSE], where TRUE is greater than FALSE.

Call By Reference -
A function or procedure call that contains an argument that can be modified by the function or procedure. Each call-by-reference argument in the function or procedure heading must have the word VAR in front of it in the argument list. An example of a call by reference is the following:

```
TYPE
    EXPRESSION = ARRAY[1..72] OF CHAR;
VAR
    STRINGIN, STRINGOUT : EXPRESSION;
PROCEDURE COUNTBLANK(VAR STRIN, STROUT : EXPRESSION);
    .
    .
    .
BEGIN
    .
    .
    .
    COUNTBLANK(STRIN, STROUT);
    .
    .
    .
END.
```

Call By Value -
A function or procedure call that contains an argument whose value cannot be modified; only a value is passed. An example of a call by value is the statement: SIN(A); The value of A remains unchanged.

CARD -
An identifier that is associated in the Pascal language with the function that returns the cardinality of the argument.

Cardinality -
Expresses how many values there are in an ordered series. For example, in the series A, B, C, D, E, the cardinality is 5.

Character Literal -
A single character enclosed in single quote (') symbols.

CHAR Type -
A simple data type that consists of the characters in the character set used at your installation. See appendix A for the available character sets.

CHR -
An identifier that is associated in the Pascal language with the function that returns the character that corresponds to the ordinal value that was passed as the argument.

CLOCK -
An identifier that is associated in the Pascal language with the function that returns the amount of CPU-time (in milliseconds) used.

Comment -
A string of explanatory text enclosed in an open comment symbol (* and a close comment symbol *). If the first character after the open comment symbol is a dollar sign $, the comment is interpreted as a list of compiler options.

Compatible -
Describes a situation in the Pascal language where the values of two identifiers can interact without conflict. For example, an identifier that describes the type of a set is compatible with the identifier that describes the type of the subrange of the set.

Conformant Array -
An array that conforms to another array. Conformant means that both arrays have one dimension and are of the same type.

Congruity -
Rules that determine whether a procedure or function can be passed as an actual parameter.

Constant -
Holds a fixed value. The value can be an identifier, enumeration constant, integer number, real number, character literal, string literal, or Boolean literal.

COS -
An identifier that is associated in the Pascal language with the function that returns the cosine of the argument.

DATE -
An identifier that is associated in the Pascal language with the function that returns the current date.

DISPOSE -
An identifier that is associated in the Pascal language with the function that releases the variable that is referenced by the argument.

Dyadic -
Describes a quantity that has two parts. For example, a dyadic function has two arguments.

Dynamic Link -
The link between data segments that enables stacking and unstacking during program execution. The word dynamic refers to the fact that the stack is constantly changing during program execution.

Enumeration Constant -
An identifier that belongs in a simple data type, except REAL type. An enumeration constant is ordered according to the placement of the identifier within the type definition.

Enumeration Type -
Another word for simple type, except REAL type. Simple data types consist of the following: BOOLEAN, CHAR, INTEGER, enumeration of a user-defined type, and subrange.

EOF (End-Of-File) -
An identifier that is associated in the Pascal language with the procedure that returns a Boolean value that depends on the position in the file.

EOLN (End-Of-Line) -
An identifier that is associated in the Pascal language with the procedure that returns a Boolean value that depends on the position in the line.

EXP -
An identifier that is associated in the Pascal language with the function that returns the exponent of the argument.

EXPO -
An identifier that is associated in the Pascal language with the function that returns the exponent of the argument in binary representation.

Expression -
A computing rule that obtains a result from applying operators to operands. An expression is evaluated from left to right using the following precedence rules:

NOT	Highest precedence
*, /, DIV, MOD, AND	↓
+, -, OR	
=, <>, <, <=, >, >=, IN	Lowest precedence

External Block -
A block that is declared in a program and defined outside of the program. An example of an external block is a function identifier that is declared with EXTERN in a program unit and is defined in the PASCLIB library.

External Reference -
A function or procedure call inside your source code to a function or procedure that resides outside your source code, for example, in the PASCLIB library. An external reference is made with the EXTERN statement.

Factor -
A part of a term and can be a constant, a variable, a function call, an expression bound by parentheses, a set value, NIL, or the complement of another factor. For example, the constant 5 is a factor.

FALSE -
An identifier that is predefined in the Pascal language as zero.

File -
Consists of a fixed number of like components called records.

File Buffer -
A template that can be positioned over any part of the file. The template isolates the part of the file that you want to read from or write to.

FILE Type -
A structured data type that describes the data on a file.

Finite -
Describes a variable that can be limited or bound.

Formal Parameter -
A variable that appears within parentheses and follows either the function identifier in a function heading or the procedure identifier in a procedure heading.

Forward Block -
A block that is declared at one point in a program and defined at a later point in the program. An example of a forward block is a function identifier that is declared with FORWARD in a program unit and is defined in a subsequent function definition.

Forward Reference -
A function or procedure call to a function or procedure that is defined later in the program. A forward reference is made with the FORWARD statement.

Function -
A block of statements that is bound by a BEGIN and an END statement. A function is called by its function identifier. The differences between a function and a procedure are that a function returns a result and a function call can be used in an expression.

GET -
An identifier that is associated in the Pascal language with the procedure that advances the position of the file to the next component.

GETSEG -
An identifier that is associated in the Pascal language with the procedure that positions the file at the start of the referenced segment counting from the current position in the file.

Global Variable -
An identifier whose scope is the entire program. A global variable can be referenced at any point in the program.

HALT -
An identifier that is associated in the Pascal language with the function that terminates the program, writes the argument in the dayfile of the job, and produces a dump.

Identifier -
A name that denotes a quantity. An identifier must be declared in the declaration part of the program unit where it is used. An identifier can be a name that denotes a constant, type, variable, value, procedure, or function. An identifier must begin with a letter followed by any combination of letters and digits up to 120 significant letters and digits.

INPUT -
A predefined identifier that is predefined in the Pascal language as TEXT. INPUT is the default textfile file name in an input textfile operation.

Integer -
An value in the range of decimal values [-2**48 + 1 .. 2**48 - 1], which is equivalent to the range of octal values [-7777777777777777 .. 7777777777777777].

INTEGER Type -
A simple data type that consists of all the integers in the range [-2**48 + 1 .. 2**48 - 1], which is equivalent to the range of octal values [-7777777777777777 .. 7777777777777777].

Internal Block -
A block that is defined and declared at the same point in a program. An example of an internal block is a function definition in the definition and declaration part of a program.

Literal -
A symbol that holds a value. There are three kinds of literals: Boolean, character, or string

LN -
An identifier that is associated in the Pascal language with the function that returns the result of applying the specified function to the argument.

Local Variable -
An identifier whose scope is the module in which it is declared and the modules that are nested within the module definition. A local variable can be referenced only within its scope.

Matrix -
A set of numbers or terms that is arranged in rows and columns.

MAXINT -
An identifier that is associated in the Pascal language with 2**48 - 1.

MESSAGE -
An identifier that is associated in the Pascal language with the function that writes the argument in the dayfile of the job.

Module -
A block of relocatable binary code that has been produced by the compiler from your source code. Each module is given a distinct address within memory by the compiler.

Modulo -
Integer division of the left operand by the right operand, the result is multiplied by the right expression, then the result is subtracted from the left expression. For example,

A MOD B is equivalent to A - ((A DIV B) * B).

Monadic -
Describes a quantity that has one part. For example, a monadic function has one argument.

NEW -
An identifier that is associated in the Pascal language with the function that allocates a new variable of the same type as the argument and assigns a reference to the argument.

Nonprinting Symbol -
Either a space or the end-of-line mark.

Object Code -
The code that the compiler produces from your source code. Also called relocatable binary code.

ODD -
An identifier that is associated in the Pascal language with the function that returns a Boolean value that depends on whether the argument is even or odd.

Operand -
An expression that is operated on by an operator. For example, in the following expression:

```
SIN(A) + SIN(B)
```

both SIN(A) and SIN(B) are operands and are operated on by the plus sign (+).

ORD -
An identifier that is associated in the Pascal language with the function that returns the number of the argument in the set of values defined by the type of the argument.

Ordinal -
A positive integer that expresses the position of a value within a series of values. For example, within the series: A, B, C, D, E, the ordinal of the value C is 3.

OUTPUT -
An identifier that is predefined in the Pascal language as TEXT. OUTPUT is the default textfile file name in an output textfile operation.

PACK -
An identifier that is associated in the Pascal language with the function that packs array values.

PAGE -
An identifier that is associated in the Pascal language with the procedure that positions the lineprinter.

Parameter -
An expression that is bound by parentheses and is operated on by a function or procedure. A parameter can consist of multiple expressions, all of which are bound by a single pair of parentheses. Also called an argument.

Peripheral -
A logical unit, such as a printer or tape drive, that is physically separate from the Central Processing Unit (CPU) but is controlled by the CPU.

Pointer Type -
A data type that describes a dynamic data structure.

PRED -
An identifier that is associated in the Pascal language with the function that returns the predecessor of the argument. If the argument is the first (smallest) value, the result may be undefined.

Predefined Word -
A name that has an associated value in the Pascal language. A predefined identifier is not a reserved word and can be redefined in the declaration part of the program unit where it is used. See section 2 for a list of predefined words and their values.

Procedure -
A block of statements that is bound by a BEGIN and an END statement. A procedure is called by its procedure identifier. The differences between a procedure and a function are that a procedure does not return a result and a procedure call cannot be used in an expression.

PUT -
An identifier that is associated in the Pascal language with the procedure that appends the value of the file buffer variable f↑ to the file f.

PUTSEG -
An identifier that is associated in the Pascal language with the procedure that closes the current segment.

READ -
An identifier that is associated in the Pascal language with the procedure that positions the referenced file and gets the referenced record.

READLN -
An identifier that is associated in the Pascal language with the procedure that gets records from the referenced textfile until an end-of-line occurs and then positions the file to the beginning of the next line.

Real -
Either a real number with an optional scale factor or a decimal integer with a scale factor. A real number is a decimal integer followed by a decimal point and up to 11 digits. A real number must be in the range [-10**322 .. -10**-293, 0, 10**-293 .. 10**322]. A decimal integer is a signed integer in the range [-2**48 + 1 .. 2**48 - 1]. A scale factor is the character E followed by a decimal integer that describes a base 10 exponent.

REAL Type -
A simple data type that consists of all the real numbers in the range [-10**322 .. -10**-293, 0, 10**-293 .. 10**322].

Record -
A collection of a fixed number of components that are called fields. A record can be divided into a fixed part and a variant part; either or both of which may be empty.

RECORD Type -
A structured data type that describes the fields in a record.

Recursive -
Describes a module that is defined in terms of itself. For example, a recursive function is a function that calls itself within the function definition.

Relocatable Binary Code -
The code that the compiler produces from your source code. Relocatable means that each block in the source code has been separated into logical records (modules) and each module has been given a distinct address within memory. Execution of relocatable binary code happens in the order that the blocks appear in the source code. Also called object code.

Repetition Factor -
An identifier that is declared in the CONST section of a program unit and holds an integer constant. For example, the repetition factor N in the following sequence initializes five elements of VECTOR to character type:

```
CONST
    N = 5;
TYPE
    VECTOR = ARRAY[1..N] OF CHAR;
```

Reserved Word -
A word that has a predefined value in the Pascal language that cannot be redefined. In this manual, reserved words are depicted in underlined uppercase letters. See section 2 for a list of reserved words.

RESET -
An identifier that is associated in the Pascal language with the procedure that positions a file to the beginning-of-information.

REWRITE -
An identifier that is associated in the Pascal language with the procedure that rewrites the referenced file at the referenced segment counting from the current position.

ROUND -
An identifier that is associated in the Pascal language with the function that returns the argument rounded to the nearest integer.

Round -
To round is to add 0.5 to the argument and then remove the decimal point and anything that follows the decimal point from the real number. Rounding yields an integer value.

Routine -
A function or procedure.

Segment -
A subdivision of a file. Segments on a file can be of varying lengths.

Scalar Type -
A data type that includes the simple and user-defined enumeration data types.

Scope -
The portion of a program over which an identifier is valid.

Separator -
A comment or nonprinting symbol. A separator can occur between any pair of consecutive Pascal symbols. A separator may appear between any pair of consecutive identifiers or literals. A separator cannot occur within a reserved word or symbol, identifier, or literal.

SET Type -
A structured data type that consists of subsets of a scalar data type.

Simple Type -
A data type that includes the following scalar data types: BOOLEAN, CHAR, INTEGER, and REAL.

SIN -
An identifier that is associated in the Pascal language with the function that returns the sine of the argument.

Source Code -
The program that you submit to the Pascal compiler.

SQR -
An identifier that is associated in the Pascal language with the function that returns the square of the argument.

SQRT -
An identifier that is associated in the Pascal language with the function that returns the square root of the argument.

Static Link -
A link between a variable and a data segment. The word static refers to the fact that the link does not change during program execution.

String Literal -
A sequence of characters enclosed in single quote (') symbols.

Structured Type -
A data type that includes the following data types: ARRAY, FILE, RECORD, and SET.

Subrange Type -
A scalar data type that consists of a subrange of another scalar data type. The subrange must be described by a minimum value, two periods, and a maximum value. The following is an example of a subrange type declaration:

```
TYPE
    DAYS = (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY);
    WEEKDAY = (MONDAY..FRIDAY);
```

SUCC -
An identifier that is associated in the Pascal language with the function that returns the successor of the argument. If the argument is the last (greatest) value, the result may be undefined.

Syntax Diagram -
A visual aid that defines a construct. In a syntax diagram, a rectangle contains the general name of a construct that you must further define, a rectangle with rounded corners contains a reserved word or symbol, and a line with an arrowhead indicates the direction of traversal within the diagram.

Term -
A part of a simple expression that can be either a single factor or multiple factors that are operated on by the following operators: *, /, DIV, MOD, or AND. For example, the constant 5 is a term. Another example, 5 * J DIV K is a term.

TEXT -
An identifier that is predefined in the Pascal language as FILE OF CHAR.

Textfile -
A file of characters. The Pascal language contains a predefined file type called TEXT, which is the same as declaring a file as FILE OF CHAR. In other words, the declaration

```
TYPE
    INFILE = TEXT;
```

is the same as the declaration

```
TYPE
    INFILE = FILE OF CHAR;
```

TIME -
An identifier that is associated in the Pascal language with the function that assigns the current time to the argument.

TRUE -
An identifier that is predefined in the Pascal language as one.

TRUNC -
An identifier that is associated in the Pascal language with the function that returns an integer whose sign is the same as the argument and whose absolute value is the greatest among the integers less than or equal to the absolute value of the argument.

Truncate -
To truncate is to remove the decimal point and anything that follows the decimal point from a real number. Truncation yields an integer value.

Type -
Describes the kind of values that a type identifier can assume. Type can be divided into the following groups: scalar, structured, and pointer.

Unary -
Describes a quantity or a function that involves one part. For example, one unary operation is the negation of one operand.

UNDEFINED -
An identifier that is associated in the Pascal language with the function that returns a Boolean value that depends on whether the value is out of range or indefinite.

UNPACK -
An identifier that is associated in the Pascal language with the function that unpacks array values.

Value -
A variable that has been both declared in a VAR declaration section and initialized with a constant, a set value, a structured value, or NIL in the VALUE section of the same declaration and definition part.

Variable -
An identifier that is associated with a value. For example, given the following declarations:

```
TYPE
    WEEKEND = FRIDAY..SUNDAY;
VAR
    WHEW : WEEKEND;
```

the variable WHEW is associated with the values FRIDAY, SATURDAY, AND SUNDAY. All variables must be declared.

Variant -
Consists of a case label list, a colon (:), and a field list enclosed in parentheses. A variant appears in the variant part of a RECORD type declaration.

WRITE -
An identifier that is associated in the Pascal language with the procedure that transforms the referenced parameter into a sequence of characters and puts the sequence onto the referenced textfile.

WRITELN -
An identifier that is associated in the Pascal language with the procedure that terminates the current line in the textfile by putting an end-of-file mark.

DIFFERENCES BETWEEN PASCAL VERSIONS 1.0 AND 1.1

D

This appendix describes the differences between Pascal Version 1.0 and Pascal Version 1.1. The differences result from conforming to the International Standards Organization (ISO) standard for the programming language Pascal. Pascal Version 1.0 is not upward compatible with Pascal Version 1.1.

SCOPE OF IDENTIFIERS

The following statements are true about the scope of Pascal Version 1.1 identifiers and not true about the scope of Pascal Version 1.0 identifiers:

- All uses of an identifier must appear after the declaration of the identifier, with one exception. The exception is that an identifier can appear as the type identifier of a pointer type before the declaration of the type identifier, as in the following example:

```
POINTER = ↑TEAMDEFN;
TEAMDEFN =  RECORD
                NUMBER : INTEGER;
                TEAMNAME : PACKED ARRAY[1..20] OF CHAR;
                TEAMMEMBERS : ARRAY[1..3] OF MEMBER;
                NEXTPNTR : POINTER
            END; (* RECORD *)
```

- An identifier in the innermost scope of a triply nested scope (a scope within a scope within the scope that defines the identifier) cannot be redefined in the next-to-innermost scope. For example, in the following sequence, the use of T in the inner record type and the use of T in the outer record type conflict:

```
TYPE
    R = RECORD
            S : RECORD
                X : ↑T              <---- This declaration conflicts
                END; (* S RECORD *)
            T : INTEGER             <---- with this declaration
        END; (* R RECORD *)
    T = REAL;
```

- The scope of an identifier extends over the entire parameter list in which it is declared and over the block of the function or procedure that corresponds to the parameter list. However, the scope does not include the function or procedure name or the result type of the function. For example, in the following sequence, the use of G as a formal parameter and the use of G as a result type do not conflict:

```
FUNCTION F (G : INTEGER) : G;      <----- The two uses of G do not conflict
BEGIN
    F := 1
END; (* F *)
```

Another example, in the following sequence, the use of P as a formal parameter and the use of P as a procedure name do not conflict. However, the use of P as a formal parameter does prevent PROCEDURE P from calling itself because P inside the procedure block can only refer to the formal parameter P.

```
PROCEDURE P (VAR P : INTEGER);
BEGIN
    P := P + 1      <----- Refers to the formal parameter P
END; (* P *)
```

See section 2 under the heading Scope Rules for the description of Pascal Version 1.1 identifier scope rules.

LABELS

The following statements are true about Pascal Version 1.1 labels and not true about Pascal version 1.0 labels:

- A label must be an unsigned integer <= 9999.
- A label that is declared must prefix a statement.

See section 2 under the heading Labels for the description of Pascal Version 1.1 labels.

FILE TYPE

The predefined type SEGTEXT has been replaced in Pascal Version 1.1 by the predefined type SEGMENTED TEXT.

See section 4 under the heading FILE Type for the description of the Pascal Version 1.1 segmented file type.

RECORD TYPE

The following statements are true about the Pascal Version 1.1 RECORD type and not true about the Pascal Version 1.0 RECORD type:

- Fields within variants can have a type that either is a file type or contains a file type.
- Variant parts of a record must be completely specified. This means that every value of the tag type must be represented by a constant prefix and a variant. For example, the type declaration RECORD CASE INTEGER OF would be impractical to use because then every integer in your program would be typed as a constant label. You can declare a subset of the integers to be typed as constant labels, as in the following sequence:

```
TYPE
    TAGTYPE = 1..3;
    RECTYPE =   RECORD CASE TAG : TAGTYPE OF
                    1 : (FIELD1 : INTEGER);
                    2 : ();
                    3 : (FIELDN : REAL)
                END; (* RECORD *)
```

- Constants that prefix a variant must lie within the tag type range. For example, the following change to the sequence above makes the record declaration invalid because the constant label 4 does not lie within the TAGTYPE range:

```
TYPE
    TAGTYPE = 1..3;
    RECTYPE =   RECORD CASE TAG : TAGTYPE OF
                    1 : (FIELD1 : INTEGER);
                    2 : ();
                    3 : (FIELDN : REAL);
                    4 : ()
                END; (* RECORD *)
```

- A tag field cannot be passed as a call-by-reference argument in a routine call; however, a tag field can be passed as a call-by-value argument.

- A tag field identifier can have the same spelling as a predefined identifier. For example:

```
TYPE
    RECTYPE =   RECORD CASE INTEGER : BOOLEAN OF
                    TRUE : ();
                    FALSE : ()
                END; (* RECORD *)
```

- Two consecutive semicolons cannot appear within a record type declaration.

See section 4 under the heading RECORD Type for the description of Pascal Version 1.1 record type.

ROUTINES

The following statements are true about Pascal Version 1.1 routines and not true about Pascal Version 1.0 routines:

- The first parameter of DISPOSE can be an expression that yields a non-NIL pointer value. For example, the first parameter can be a function whose value is a pointer.

- HEX for integer output to textfiles is not allowed. To write integers in hexadecimal, you can use the following declarations and statement:

```
(*$X0 PASS NO PARAMETERS IN X REGISTERS *)
PROCEDURE (*$E'P.WRH'*) WRITEHEX (VAR F : TEXT; I,W : INTEGER);
    EXTERN;
(*$X= RESTORE X-OPTION *)

WRITEHEX(file, integer, fieldwidth);
```

- HIGH has been eliminated.

- LINELIMIT has been replaced by the PL compile command parameter.

- LOW has been eliminated.

- The first parameter of NEW can be a component of a packed array.

- ORD(sets) has been eliminated.

- OCT for integer output to textfiles has been eliminated. To write integers in octal, you can use the following declarations and statement:

```
(*$X0 PASS NO PARAMETERS IN X REGISTERS *)
PROCEDURE (*$E'P.WRO'*) WRITEOCT (VAR F : TEXT; I,W : INTEGER);
    EXTERN;
(*$X= RESTORE X-OPTION *)

WRITEOCT(file, integer, fieldwidth);
```

- The following statements are true about PACK and UNPACK:

  The element types of the packed and unpacked arrays must be identical.

  The index types of the packed and unpacked arrays can be incompatible.

  The routines can be used on arrays that are declared PACKED but are not actually packed in memory. An array is not packed in memory when individual elements occupy one or more words of storage.

- The following statements are true about PAGE:

  A parameter is not required; the default file parameter is OUTPUT.

  A WRITELN(f) is not implicitly performed unless the last operation on file f left a partial line. For example:

```
BEGIN
    REWRITE(f);
    PAGE(f);      <----- A WRITELN(f) is not performed
    WRITELN(f,' HELLO ');
    PAGE(f);      <----- A WRITELN(f) is not performed
    WRITE(f,' PARTIAL LINE ');
    PAGE(f)       <----- A WRITELN(f) is performed before the
END.                     page-eject
```

- The following statements are true about READ:

  The variable arguments (v1, ..., vn) can be components of a packed array or packed record.

  Execution terminates when the end-of-file condition becomes TRUE after skipping spaces while reading from file f (under Pascal Version 1.0, an undefined value message is issued when the end-of-file condition becomes TRUE after skipping spaces while reading from file f).

- RELEASE has been eliminated.

- The following statements are true about WRITE:

  A Boolean expression in a WRITE procedure call is equivalent to either of the strings 'TRUE' or 'FALSE' in the WRITE procedure call.

  WRITE(pointers) and WRITELN(pointers) is not allowed; use WRITE(ORD(pointer)) or WRITELN(ORD(pointer)).

  Execution terminates when a zero or negative format width is encountered while writing a variable argument (v1, ..., vn). For example, the following WRITE procedure calls are not valid:

```
WRITE(OUTPUT,NUMBEROFITEMS:0);
WRITE(DATA,VARIANCE:10:0);
WRITE(CLASSDATA,STUDENTS:-1);
```

  Execution terminates when an undefined variable is encountered while writing to file f (under Pascal Version 1.0, an undefined value message is issued when an undefined variable is encountered while writing to file f).

See section 5 under the appropriate routine heading for the description about the Pascal Version 1.1 routine.

PARAMETER LIST CONGRUITY

The parameter list of a procedure or function must be congruous with the parameter list of the procedure or function call.

See section 5 under the heading Procedure and Function Parameters for the description of Pascal Version 1.1 parameter list congruity.

CONFORMANT ARRAY PARAMETERS

Conformant array parameters that correspond to formal parameters in the PROCEDURE section can share a common conformant array descriptor. For example:

```
PROGRAM MAIN;
TYPE
    VECTOR1 = ARRAY[1..100] OF REAL;
    VECTOR2 = ARRAY[1..50] OF REAL;
VAR
    A, B : VECTOR1;
    C, D : VECTOR2;
PROCEDURE P1 (X,Y : ARRAY[LO..HI : INTEGER] OF REAL);
BEGIN
  .
  .
  .
END; (* P1 *)
PROCEDURE P2 (X : ARRAY[LO1..HI1 : INTEGER] OF REAL;

Y : ARRAY[LO2..HI2 : INTEGER] OF REAL);
BEGIN
  .
  .
  .
END; (* P2 *)
BEGIN (* MAIN *)
    P1(A,B);
    P1(C,D);
    P2(A,C);            Note that calls to P1(A,C) and P1(B,D) would
    P2(B,D);            be invalid because of incongruity.
    P2(A,B);
    P2(B,C);
    P2(D,A);
END. (* MAIN *)
```

See section 5 under the heading Parameters for the description of Pascal Version 1.1 formal parameters.

ASSIGNMENTS TO FUNCTIONS

An assignment to a function variable must occur within the function block. In Pascal Version 1.1, the assignment can occur within a nested procedure or function. For example:

```
FUNCTION OUTER (J : INTEGER) : INTEGER;
PROCEDURE INNER (K : INTEGER);
BEGIN
    IF K >= 10 THEN OUTER := K
    ELSE OUTER := 10
END; (* INNER *)
BEGIN
    INNER(J)
END; (* OUTER *)
```

FOR STATEMENT

The following statements are true about Pascal Version 1.1 FOR statement control variables and not true about Pascal Version 1.0 FOR statement control variables:

- A control variable must be a local variable.
- A control variable cannot be a formal parameter.
- A control variable cannot be assigned a value.
- A control variable cannot be used as the control variable in a nested FOR statement block.
- A control variable cannot be used as an actual parameter in a routine called in the FOR statement block.
- A control variable cannot be used as a parameter in a READ or READLN procedure.
- A control variable that has been defined with an initial and final value must lie within the initial and final value. For example, in the following sequence, the FOR statement is invalid because the initial value of I is not within the range defined for I:

```
VAR
    I : 10..20;
BEGIN
    FOR I := 1 TO 20 DO A := A + 1;
```

See section 6 under the heading FOR Statement for the description of the Pascal Version 1.1 FOR statement.

GOTO STATEMENT

The Pascal Version 1.1 GOTO statement can contain a label only if one or more of the following conditions is true:

- The label prefixes a statement that contains the GOTO statement. For example:

```
1 : IF A THEN PROCA
    ELSE BEGIN
        PROCB;
        GOTO 1
    END; (* ELSE *)
```

- The label prefixes a statement in a statement sequence and another statement in the sequence contains the GOTO statement. For example:

```
    REPEAT
1 :     PROCA;
        IF (A < B) THEN BEGIN
            PROCB;
            GOTO 1
        END; (* IF *)
        PROCC
    UNTIL DONE
```

- The label prefixes an unnested statement in a procedure or function block and the GOTO statement occurs in another procedure or function block that is nested within the procedure or function. For example,

```
PROCEDURE OUTER;
LABEL
    1;
PROCEDURE SANCTUM;
BEGIN
    IF (A < B) THEN GOTO 1
END; (* SANCTUM *)
BEGIN
    INITIALIZE;
    SANCTUM;
    IF (C < D) THEN BEGIN
      .                      The label cannot prefix a statement inside
      .       <------------- the IF statement sequence because it would
      .                      not prefix an unnested statement.
    END; (* IF *)
1 : FINALIZE
END; (* OUTER *)
```

See section 6 under the heading GOTO Statement for the description of the Pascal Version 1.1 GOTO statement.

COMPILE COMMAND

The parameters on the Pascal Version 1.1 compile command are order-independent; however, all options must follow the parameters and the slash (/).

Table D-1 shows how the compile command options have changed:

Table D-1. Compile Command Option Changes

Pascal Version 1.0 Option	Equivalent Pascal Version 1.1 Option
G+	GO
L-	L- or L=0
Ln	Either PD or PS
-	O
-	PL
-	S

See section 7 under the heading Compiling a Program for the description of the Pascal Version 1.1 compile command.

INDEX

COMMENT SHEET

MANUAL TITLE: Pascal Version 1.1 Reference Manual

PUBLICATION NO.: 60497700

REVISION: A

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments on the back (please include page number references).

_____ Please reply _____ No reply necessary

FOLD FOLD

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division
P.O. BOX 3492
Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD FOLD

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.
FOLD ON DOTTED LINES AND TAPE

NAME:

COMPANY:

STREET ADDRESS:

CITY/STATE/ZIP:

TAPE TAPE

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION